The Journal of Systems and Software 83 (2010) 1780–1800

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A software integration approach for designing and assessing dependable embedded systems

Neeraj Suri[a,*], Arshad Jhumka[b], Martin Hiller[c], András Pataricza[d], Shariful Islam[a], Constantin Sârbu[a]

[a] TU Darmstadt, Germany
[b] University of Warwick, UK
[c] Volvo Technology Corporation, Sweden
[d] Budapest University of Technology and Economics, Hungary

ARTICLE INFO

Article history:
Received 31 October 2008
Received in revised form 21 April 2010
Accepted 22 April 2010
Available online 15 June 2010

Keywords:
Embedded systems
Dependability
Software integration
Assessment
Decision theory

ABSTRACT

Embedded systems increasingly entail complex issues of hardware–software (HW–SW) co-design. As the number and range of SW functional components typically exceed the finite HW resources, a common approach is that of *resource sharing* (i.e., the deployment of diverse SW functionalities onto the same HW resources). Consequently, to result in a meaningful co-design solution, one needs to factor the issues of processing capability, power, communication bandwidth, precedence relations, real-time deadlines, space, and cost. As SW functions of diverse criticality (e.g. brake control and infotainment functions) get integrated, an explicit integration requirement need is to carefully plan resource sharing such that faults in low-criticality functions do not affect higher-criticality functions.

On this background, the main contribution of this paper is a dependability-driven framework that helps to conduct the integration of SW components onto HW resources such that the maintenance of system dependability over integration of diverse criticality components is assured by design.

We first develop a clustering strategy for SW components into Fault Containment Modules (FCMs) such that error propagation via interaction is minimized. Subsequently, the rules of composition for FCMs with respect to error propagation are developed. To allocate the resulting FCMs to the existing HW resources we provide several heuristics, each optimizing particular attributes thereof. Further, a framework for assessing the *goodness* of the achieved HW–SW composition as a dependable embedded system is presented. Two new techniques for quantifying the goodness of the proposed mappings are introduced by examples, both based on a multi-criteria decision theoretic approach.

1. Introduction and problem perspectives

For embedded systems, ranging from service-critical cell phones to safety-critical aerospace/automotive control, their functional and extra-functional attributes (e.g., performance, dependability; Avizienis et al., 2004, timeliness, power, etc.) are increasingly defined by the *software components* (SW-Cs). Given the drivers for low cost, low power, shorter development time and to accommodate continuing functional upgrades, the trend is to move from the traditional *federated* (Rushby, 1999) to *integrated* embedded architectures. In the federated paradigm each SW function is allocated to its own HW node, while in the integrated architecture several SW functions (having various requirements) share a set of common HW nodes.

While a federated approach using dedicated and partitioned-by-design hardware and separate networks for each function is desirable for cleanly partitioned functionality and fault-tolerance, realistically it is prohibitive expensive from a resource (power, space, weight, development cost, etc.) consideration. Consequently, integrated approaches (such as the initial IMA approach of Lee et al., 2000; Younis et al., 2004; Islam et al., 2009, etc.) are often advocated for integrating diverse software components (SW-C) onto shared HW resources for cost reasons. The SW-Cs differ not only in functionality, but also in extra-functional requirements (e.g., criticality, dependability, timeliness). For example, the integration for flight control system involves placing display, sensor, collision avoidance, and navigation SW-C onto a shared HW platform[1] – the plane's main computer. Hence integrating mixed-criticality SW-C's requires careful attention that

* Corresponding author at: TU Darmstadt, Computer Science, Hochschulstr. 10/S2|02 E217, Darmstadt, Germany. Tel.: +49 6151 16 3513; fax: +49 6151 164310.
E-mail address: suri@cs.tu-darmstadt.de (N. Suri).

[1] E.g., Airplane Information Management System (AIMS) in the Boeing 777 addresses composite information management functions.

0164-1212/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.04.063

their fault-tolerance (FT) and real-time (RT) requirements are not compromised with proper fault-containment assured across the integrated system.

As the diverse SW-Cs interact in the spatial and temporal dimensions, their access to shared resources along these two dimensions must also be carefully controlled. To meet this objective, Rushby (1999) advocated a design concept called *p* artitioning, which calls for well-defined boundaries across modules to ensure the continuity of operation in the presence of errors. The goal of partitioning is to create error containment units such that the behavior of one partition is unaffected by the (faulty) behavior in another partition.

In this work, we utilize the basic models of partitioning and IMA (Lee et al., 2000) integration, to develop a powerful quantifiable notion of partitioning, termed as *influence* and *separation*, for designing an integrated embedded system. Influence and separation, respectively quantify the degree of linkage or distance across SW-C's. In general it is needed that the safety-critical SW-Cs are shielded from lower criticality SW-Cs, but not vice-versa. For example, it might be acceptable (in terms of safety) for a flight control subsystem to corrupt the entertainment subsystem in a plane, but there will be catastrophic consequences if the temperature measure subsystem corrupts the collision avoidance subsystem. In such a context, partitioning is equivalent to maximum separation or minimum influence.

Using the key concepts of influence and separation, we develop a systematic integration framework for (a) constructing SW-C's and (b) their mapping onto the targeted HW platform, specifically for designing integrated dependable real-time embedded systems. Such a framework explicitly needs to support dependability-aware specifications and integration, and provide assurance for the correctness of design and implementation. Additional issues include supporting SW evolution, reuse, and cross-platform portability.

With dependability being our primary design objective, the proposed integration framework emphasizes the handling of faults in SW, and, where possible, on minimizing their number, scope, and their effects. In the HW arena, an established approach is to create *fault containment regions* [2] at and across architectural, information flow, and timing levels. We seek an analogous approach for SW partitioning, utilizing established solutions as replication and design diversity with additional coverage for SW/HW interactions to specifically provide for:

- **Ensuring the desired level of non-interference** below a threshold influence level, of operation between SW modules, and providing effective guidelines for support of non-interference. However, for high critical processes executing on the same processor there is always a need of partitioning within each processor or processor core to restrict spatial and temporal interactions.
- **Delimiting the scope of a fault**, restricting the possible sites for correlated[3] faults. In contrast to the fault containment regions approach for HW, the SW approach must consider aspects such as process migration, memory sharing, and inter-function logical and temporal dependencies.
- **Selecting an appropriate level of integration** to design for, and ascertaining and quantifying trade-offs involved over the integration process. A key aspect described in this paper is the assessment of the mapping process.

In the next section we outline the basics of our approach and subsequently describe the various strategies contained therein in the following sections.

1.1. Approach and our contributions

The composition of dependable SW is often a complex process involving conflicting demands of performance and reliability to be handled concurrently. Compositionally, it is desirable to have software partitioned into *Fault Containment Modules* (FCMs)[4] with associated characteristics and having these FCMs interacting in a prescribed manner. Given such a partitioning, systematic composition of partitions into an integrated design simplifies the study of error containment over an evolving design. In (Jhumka et al., 2002c), we developed a framework that provides guidelines for composition of FCM partitions, and in (Jhumka et al., 2002b), we extended the framework for test case generation to ascertain the safe composition of those partitions.

This paper proposes a framework for the synthesis of dependable SW allowing for integration of SW-Cs associated with attributes such as criticality, timeliness and reliability. The paper specifically addresses the dependability-driven mapping (focuses on minimizing influences across modules) and presents some heuristics for realizing viable mappings. A mapping is defined as (i) assigning different SW-Cs to suitable HW nodes such that platform resource constraints and dependability requirements are met (*resource allocation*) and (ii) ordering process executions in time (*scheduling*) (Islam et al., 2006). After the integration processes, once a solution or a set of contending solutions are obtained, the quality of the obtained solution(s) need to be evaluated against system requirements. Therefore, sound methods for assessing the *goodness* of each obtained mapping are required to select the best one from the contenders. We propose rigorous and systematic approaches for assessing *good* designs and selecting the near-optimal solution.

Accordingly, this paper makes the following contributions:

(1) **A dependability-driven design methodology for distributed embedded systems**, where composition of smaller entities into basic FCMs, mapping of FCM components onto HW platform and the mapping assessment process are the key properties. Dependability is ensured through replication of high critical SW-Cs. Once a partitioned FCM structure is obtained, it needs to be allocated onto HW nodes satisfying a set of constraints. We also describe the technique of reducing the influence between different modules thereby maximizing dependability. To this end, a set of heuristics optimizing desired dependability features are developed.
(2) **An approach to transform standard object-oriented designs into hierarchical FCM structures**. As the application may not have been designed with dependability as overriding goal, there may be a need to transform a given SW module into an FCM, which may be achieved through techniques such as Error Detection Mechanisms (EDMs), Executable Assertions (EAs) (Hiller, 2000; Saib, 1978) and Error Recovery Mechanisms (ERMs).
(3) **Quantified inter-module interaction**. Since unwanted interactions between different modules may exist, we quantify the respective interactions among them, and cluster them such that error propagation is minimized, while SW- and HW-constraints are respected.
(4) **A set of rules allowing composition of basic FCMs into reliable hierarchical FCM structures**. The transformation of a

[2] A *fault containment region* represents the system boundary where the effect of a fault is to be contained.

[3] The failure of a module due to the failure of another module is called a *cascading failure*.

[4] FCMs are, at a general level, analogous to the *fault containment regions* used in a HW context. These are discussed in Section 3.

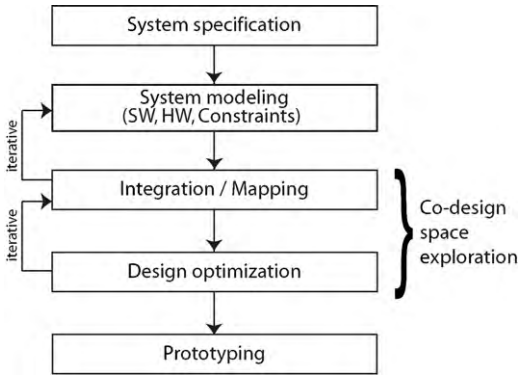

Fig. 1. Abstract view of system level design steps.

standard object-oriented design, implementation, and maintenance into a reliable FCM structure is subject to specified constraints on the language features, data structures, and design features used in the application.

(5) **Two approaches for assessing the "*goodness*" of HW–SW mappings**. They account for the constraints imposed by the SW-Cs as well as by the HW platform and system preferences. The first approach is a search-based technique called as "*random exploratory technique*", yielding a true/false result, depending on whether the mapping satisfies the system requirements. The second approach strengthen the first one by introducing a 9-step interactive decision procedure that systematically guides in selecting the *best* available mapping.

Overall, *we envision the process to systematically guide the design and development of SW and its mapping for dependable RT embedded systems.* However, this can also be applied as a post-pass to existing embedded system designs.

This systematic process' steps are depicted in Fig. 1. The system level design is started by specifying the system and constructing the HW and SW models separately out of the defined specification. Then the SW components are mapped onto the available HW resources. The mapping is decomposed into two subproblems: *allocation* and *scheduling*. The design flow includes activities such as system specification, SW, HW and constraints modeling, dependability- and real-time-driven SW–HW integration, design optimization and the prototyping of the proposed design. Some design steps are performed and refined iteratively (i.e., finding a mapping and its optimization). The co-design space optimization enables the exploit of search spaces by investigating alternative mappings and their optimizations to be able to find an (near-)optimal mapping from the global design space. All these steps are detailed onwards in the different sections of this paper.

1.2. Paper organization

Section 2 starts by presenting the related work in the area of software integration. Then, Section 3 presents the system model, the constraints model and the fault model assumed in this work. Section 4 presents the techniques for defining SW FCMs. The integration rules for SW FCMs are introduced in Section 5, which also describes the measures of *influence* and *separation* among modules. Section 6 describes general aspects of the mapping process and also a set of heuristics for realizing the mappings, each optimizing a certain criteria. An example of a real mapping is given in Section 7.

In order to quantitatively assess the *goodness* of each heuristic-driven mapping, Section 8 introduces the utility of decision theory for integration, and also discusses the motivation and relevance of a goodness function. Section 9 then expands on the presented example and applies a decision theory-based framework to determine the goodness of the different mappings to select the best available one. Section 10 summarizes the decision procedure and provides some insights on the procedure and on some of its limitations. Section 11 concludes the paper.

2. Related work in software integration

Varied techniques have already been used for solving the resource allocation problem, e.g., constraint propagation (Ekelin and Jonsson, 2001; Kuchcinski, 2003), inform branch-and-bound and forward checking (Kuchcinski, 2003; Wang et al., 2004) and mixed integer programming (Rajkumar et al., 1998). These approaches typically perform the mapping (allocation and scheduling) straightforwardly applying the above mentioned techniques. A disadvantage of these approaches is that usually they do not put additional efforts to reduce the search space a priori while solving the problem thus limiting their applicability to handle only a few constraints. (Ekelin and Jonsson, 2001) applies symmetries exclusion to reduce the search space which is more desirable in a homogeneous system. An enhancement of the Quality-of-Service (QoS) based resource allocation model (Rajkumar et al., 1998) is presented in (Ghosh et al., 2003), where a hierarchical decomposed scheme by dealing with smaller number of resources is described enabling QoS optimization techniques for large problems. Tasks replication is used as a QoS dimension in order to provide fault tolerance.

The major requirements for designing embedded systems are to meet both RT requirements and to provide dependability (FT, avoiding error propagation, etc.). Commonly used approaches typically address RT and FT on a discrete basis (Wang et al., 2004; Ghosh et al., 2003). AIRES (Automatic Integration of Reusable Embedded Systems) (Wang et al., 2004; Kodase et al., 2003) describe the allocation of SW components onto HW platforms for RT and embedded applications satisfying multiple resource constraints. They also provide a schedulability analysis. The method has been implemented into a Model Driven Development (MDD) analysis tool that evaluates whether those constraints are satisfied. Based on constraint programming, (Kuchcinski, 2003) presents an approach to constraint-driven scheduling and resource assignment. They develop a constraint solver engine which satisfies a set of constraints. However dependability/FT is not considered in any of these approaches. Moreover when scheduling for RT systems is performed, a predetermined allocation or a simple allocation scheme is used (e.g., Lee et al., 2000). If the scheduling is performed without assuming any pre-allocation it may significantly increase the computation complexity and can make the problem intractable (cannot be solved in polynomial time; Garey and Johnson, 1979). Also if the allocation and scheduling are considered separately, important information (e.g., considering constraints) used from one of these activities is missed while performing the other. On the other hand, usually FT is applied to an existing scheduling principle such as rate monotonic or static off-line either by using task replication (Oh and Son, 1994) or task re-execution (Kandasamy et al., 1999). Existing approaches typically do not address all the constraints or use a limited fault model where dependability is essential. Suri et al. (1998) specifically addresses the dependability driven mapping (focuses on minimizing interaction) and presents the heuristics for doing the mapping. However the focus is on design stage SW objects to aid integration. A survey of various SW development processes addressing dependability as extra-functional requirements at both late and early phases is described in Mustafiz and Kienzle (2004). Yin et al. (2006) provides a tool suite for the design and analysis of large-scale embedded RT systems.

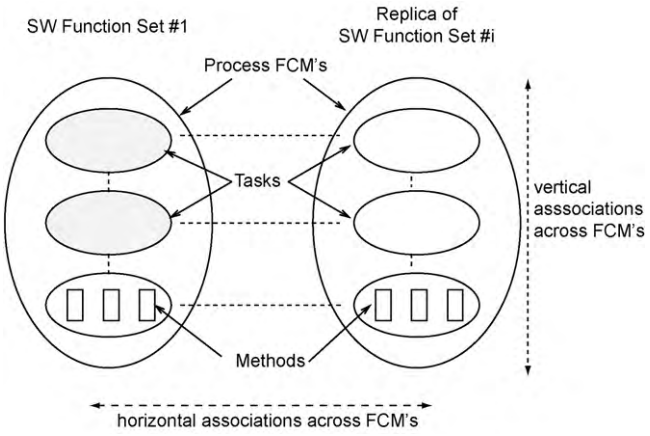

Fig. 2. The SW model: SW (FCMs) layers across (replicated) SW-Cs.

The DESERT tool suite (Mohanty et al., 2002; Neema et al., 2003) utilizes a generic, multi-level abstraction framework for the design space exploration problem. It supports the modeling of design spaces and the automated search for design that meet the requirements. The space explosion problem is handled by using iterative refinement techniques starting from an abstract problem model towards detailed, implementation level solutions. The tool can be integrated into existing development and analysis tools as a generic solver engine. DESERT has been used in several projects in the automotive domain.

Embedded virtualization (Baldin and Kerstan, 2009; Wind River, 2010) is an emerging concept of the industry, inspired by the success of server and desktop virtualization solutions. Its key goals are similar to that of the partitioning, namely the spatial and temporal separation of software components sharing the same hardware resources, moreover, it enables the utilization of different operating systems in the virtual machines allowing the integration of subsystems of mixed criticality while maintaining the error containment regions. With the increasing capacity (and core count) of embedded processors, the management of computing resources becomes also an important benefit of virtualization. Several academic and industrial implementations of the technology are already available.

3. System model

We utilize a system model partitioned into the constituent SW and HW models, the functional and extra-functional constraints model, and the fault model. For ease of presenting the constitution of SW FCM, the SW model is decomposed into a multi-level model. The HW platform is the execution environment of SW-Cs, whereas the constraints model defines a set of constraints that need to be satisfied during the mapping. The fault model describes an anticipated set of both SW and HW faults. In the following we detail all applied models.

3.1. The SW model

We utilize a four-level SW model: *processes*, *tasks*, *o* bjects and *methods* (cf. Fig. 2). This choice of a four-level model is used to conform to conventional SW programming models. In reality, the user can choose any *n*-level model, depending on the needed degree of abstraction. We assume a system consisting of multiple processes, with shared-memory- or message-passing-based communication, executing in a multi-node/OS environment.

In the SW hierarchy presented in Fig. 2, a *process* represents a heavyweight thread of control. Each process consists of a set of single-threaded *tasks*, each with a separate conceptual code and data space (most likely with physical overlays, at least in cache), and a private program counter and stack. Processes may send messages which use, reserve, or release resources (e.g., for performing input/output-related operations). At this level, we consider a process to have the following properties: (i) process name – each process has a unique name; (ii) timing requirements – earliest start time *EST*, computation time *CT*, deadline *D*; (iii) bandwidth requirements for inter process communication; (iv) criticality requirements – reflecting how critical a process is; and (v) dependability requirements – the degree of replication dc_i for *i*th process as needed to provide the required level of FT.

Processes of a function can be modeled as a weighted directed acyclic graph, where vertices represent the properties of the processes and the edges between processes represent the value of the error propagation probabilities.

Tasks are lightweight threads of control. Each task consists of a set of *objects*, with calls only within a task or from task procedures to per-processor replicated system utilities with known behavior. Tasks within the same process communicate via messages. An *object* may not possess an independent thread of control. It consists of a data space and a set of functions called *methods* that describe the behavior of the object. Each object is a named SW-C callable through its methods. Related objects belong to the same task. A *method* is likewise a named and callable SW-C, it can be a procedure or a function. A *method* does not have a thread of control. We detail these SW stratifications further in Section 4.

3.2. The HW model

While we envision our approach to aid in system level SW/HW co-design, this paper initially considers a basic HW inter-node topology with all nodes able to communicate with each other, e.g., as commonly applied in automotive and avionic control systems. A HW node is a self-contained computational element (single- or multiprocessor) connected to the network (e.g., a *bus* topology) with a communication controller along with additional resources, e.g., sensors, actuators, etc. The communication controller manages the exchange of messages with other nodes. We assume node processors to have access to equivalent sets of resources.

3.3. The constraints model

The constraints define the conditions that limit the possible mappings from a dependability, real-time or resource perspective. A set of constraints need to be satisfied for a mapping to be valid (Islam et al., 2006). We define the following types of constraints: (i) binding constraints – processes that need to be allocated onto specific nodes due to the need of certain resources (e.g., sensors or actuators). This constraint can be referred to as *type matching* between the processes requiring sensors/actuators and the HW nodes attached with sensors/actuators; (ii) dependability constraints – separation of replicas to different nodes; (iii) schedulability constraints – maintaining timing requirements and (iv) computing constraints – available resources, such as the amount of memory available for processes.

3.4. The fault model

The occurrence of faults is considered (a) within an FCM and (b) over communication across them. The consequence of a fault is an error (deviances from the functional or temporal specifications) which can propagate from a source module to a target module explicitly via an erroneous message sent by a faulty job or implicitly via some shared resource. Fault/error occurrence and transmission probabilities are taken to be independent of the locations of the source/target processes. We disregard indirect error interactions

arising from 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. Specifically, fault occurrences within a FCM can be approximated using field data, or using mathematical modeling tools, such as stochastic modeling, or extreme value theory (Kaufman et al., 2002). Also, direct or indirect error transmission probabilities can be approximated using the methods presented in Section 5.3 and in Section 5.4, where we develop the *i* nfluence, and *separation* metrics for direct and indirect fault propagation respectively. For HW faults, we assume both transient (the types of faults which appear for a short duration of time) and permanent faults (the types of faults which may causes permanent damage to a component).

4. SW FCMs at different levels: an overview

As we deal with white-box SW, we first endeavor to characterize the behavior of SW FCM at each level (Fig. 2). A given SW FCM, at a given hierarchy level, is created by establishing isolations from other FCMs it might interact with are clearly isolated from it, thus satisfying restrictions on how FCMs at a given level can interact. Each FCM delineates the specific classes of errors at each FCM layer. For example, at the process level, two processes can be assigned to disjoint memory areas; at the method level, techniques such as executable assertions (EAs) can be used to check the validity of data being passed between procedures. Once an FCM is created, verification tests are conducted to ensure that its interactions with other FCMs do not violate any overall SW/FCM specifications, for example, EAs placed in one module being defeated by subtle omissions of other EAs placed in other cooperating FCMs (Jhumka et al., 2002a), i.e., whether the EAs are consistent with each other.

In this paper, our notion of FCMs corresponds to some well-known programming language paradigms, such as objects, processes, etc. However, depending on the level of detail needed, the system designer may increase or decrease the number of levels in the hierarchy. For example, if the system designer feels that if a fault occurs at the method level, and that fault can be masked at the object level in such a way that one does not need to focus on the method level, then such an abstraction can be used, i.e., 3-layers instead of the proposed 4-layers. This analogously applies to HW–SW interactions. To illustrate this, consider the following example: a fault occurs at the HW level (for example, in a transistor). This results in an error at the gate level, and the fault also manifests itself as an error in the SW layer. Depending on the abstraction required, there are two possibilities, (i) either the error is handled at the gate level, or (ii) at the SW level. If the error is handled at the SW level, then one does not need to focus on the HW level. Thus, we focus on higher level FCMs, rather than low level FCMs. For each FCM level, we outline the requisite error classes as follows:

- **Process-level (top level) FCM:** The *process* level FCM is the topmost level in the hierarchy of Fig. 2, and represents a heavyweight (e.g., UNIX-like) process. Each such *process* has its own code and data, plus associated characteristics such as criticality and timing constraints. Faults and errors to be handled at this level arise from sharing HW resources. Examples include memory space overlapping (*m* emory footprints), timing, scheduling and communication faults. The communication can take place in two different ways (intra-process and inter-process) depending on the placement of processes. If the communicating processes are located on the same processor it is termed intra-process communication and inter-process communication otherwise. The error propagation is considered at the inter-process communication level.

  Techniques for constraining fault/error scope are required to ensure that a fault or error within one process does not lead to a correlated fault/error in another. This may require processes to be shielded, both spatially and temporally, using techniques like separating the memory blocks used for their execution, or ensuring against overuse of resources, (e.g., CPU) using techniques such as the one described in (Lipari et al., 2000).
- **Task-level FCM:** The *task* level FCM is the second level in the hierarchy. Tasks are lightweight threads that can share data and memory, each with its own stack and program counter. A group of tasks which share data and text belong to the same *process* FCM.

  At *task* level, faults occurring in one task may affect other tasks within the same *process* FCM. For example, 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 can be applied for error containment (e.g., N-version programming or Recovery Blocks) and error detection (e.g., EAs for data errors; Hiller, 2000; Saib, 1978. In Hiller et al. (2004), the concepts of error permeability of tasks were introduced. Informally, the permeability metric of a task is an indication of the relative ease with which that task allows errors to propagate. To constrain error propagation, error handling mechanisms, such as replication, or error detection and recovery mechanisms, can be used. As always, replication is an expensive alternative. Consistency protocols are needed to keep the tasks consistent. Thus, as advocated in Hiller et al. (2004), adding software mechanisms, such as EAs provides a cost-effective way to detect (and subsequently contain) errors. If two *process* level FCMs need to communicate, they spawn *task* level FCMs within the same *process*, (as in socket-based communication). Thus, communication level faults get addressed only at *task* level.
- **Object-level FCM:** The *object* level is the third level in the software hierarchy of Fig. 1. An object is a dynamic entity (exists at run-time), and is an instance of a class. An object is a named and callable SW entity. A group of functionally-related objects with associated characteristics form a task. For example, a temperature sensor in a furnace reads the temperature, passes it on to another object, which does some processing with this input. Then, an appropriate message is sent to another object for action. As these three objects are closely coupled, they are merged to form a task. Objects are characterized by a set of attributes, which is divided into two distinct categories of (a) *operations*, and (b) *data attributes*.

  There are three types of objects, namely *actors*, *agents* and *servers*. Actors always make calls to other objects and their methods are never externally invoked. Servers service requests passed onto them and never make any method calls. Agents both send messages to other objects and service requests passed onto them.

  One type of fault to be tolerated at this level is the passing of erroneous data through the object's methods via Remote Method Invocation (RMI), i.e., an object sending an erroneous message to another object.
- **Method-level FCM:** A method is an object attribute that will either query or modify the state of that object. It is a named and callable SW entity. Making a remote call to a method is semantically equivalent to sending a message to the object to which this method belongs.

  Methods (procedures) are categorized as either being *risky*, if they are parameterized, or *safe*, if they are not, since errors to be tolerated at this level manifest themselves as erroneous parameter values. Possible techniques to contain errors are use

of redundant inputs or EAs to monitor validity of the input values. Another type of error may be related to control flow, which can then be taken care of through, e.g., signature checking.

5. Basic integration of SW FCMs

The realization of an integrated system using this approach is a two-phase process. The first step entails decomposition (and possible transformation) of a SW function into FCMs and their systematic – vertical and horizontal – integration. In this section we describe these aspects. The second step involves assigning the SW FCMs to processors satisfying the defined constraints, a step which is described in Section 6.

The dependability-driven integration problem takes a set of SW modules, most likely functionally clustered, and partitions them into different hierarchical FCMs. As this functional decomposition is not designed for fault isolation, transformations might be required to obtain a final procedure and task decomposition into FCMs. Also, FT requirements may require adding redundancy at any level of the FCM hierarchy. So far we have expanded the SW modules to demarcate their exact operations, and to identify their interactions. As our goal is the integration of SW-Cs, we now develop techniques for systematic assimilation of the obtained FCM modules. As the dependability of SW is our key integration criteria, we now propose *rules* for composing the SW FCMs of Fig. 2 to ensure that errors are not propagated but contained and tolerated, with a specified and quantifiable degree of confidence. The importance of constructing these rules relies mainly on confining the fault/error propagation between different hierarchy layers of SW FCMs (see Fig. 2). The introduction of these composition rules for the integration does not add additional complexity as well as does not introduce additional constraints in the process. We do not also restrict the application scope.

We consider two kinds of SW integration: *Vertical (Global) integration* and *Horizontal (Local) integration*. Vertical integration addresses systematic composition of locally consolidated FCMs into an FCM at the next higher level of hierarchy (hence global). *Vertical* integration is thus hierarchical, integrating SW FCMs at one level into larger FCMs within the defined FCM hierarchy, e.g., combining methods to form an object.

Alternatively, horizontal integration addresses interactions across locally consolidated FCMs. The intent is to understand interactions and error propagation across SW-Cs within a parent SW (hence local) that are desired to be integrated in the embedded environment. Modules are combined by grouping, i.e., the FCMs retain their respective interfaces. Grouping, in an object-oriented context, may be achieved via inheritance where more *specialized* classes (objects) are created.

5.1. Vertical (Global) integration of FCMs within a SW component

Vertical integration involves FCMs at different levels of FCM hierarchy within a specified SW-C. The clustering of FCMs at one level into FCMs at a higher level needs to satisfy the following rules:

R1a: *Any number of FCMs at one level can be integrated to form an FCM at the next higher level.*

For example, one or more *methods* can be integrated to create an *object*, and one or more *objects* to create a *task*. This creates a layered *integration DAG* (Directed Acyclic Graph). We will use the terms *parent*, *child*, and *s* ibling in this context.

R1b: *Each task should possess, at least, either an actor with a periodic thread of control or an aperiodic one though not a random one. They should also possess at least an agent object as well as a server object.*

This ensures that *tasks* fundamentally cooperate to obtain results. It also provides a certain guideline as to which FCMs can be integrated with each other. The reason why we cannot have only agents is that none of the agents may possess a thread of control, for which we have the actor object. Thus, having actors, and agents are necessary. Having server objects is only necessary when none of the agents is providing a given service. However, since we are interested in transforming an existing object-oriented design, initial verification performed should have already verified the need for server objects. For transformation purposes, no additional server object is needed.

R2: *The integration DAG is a tree.*

This rule allows the fault containment boundary of FCMs to be well-defined. Also, this rule does not hamper reuse since an object is a dynamic concept. Thus, even when two different tasks need to access the same object (class), two separate copies (instances) can be created. The code size does not increase, hence does not decrease the efficiency of code generation. Also, if two separate tasks need to cooperate by sharing a common object (i.e., having pointers to the same object in memory), thus violating this rule, it can be circumvented by using the *Clone* method, for example as in Java (Flanagan, 1999), creating identical copies for each task. Creating identical cloned copies creates the impression of having consistent objects. However, since each copy belongs to a given task, then no cross-boundary access is needed, thus enhancing fault containment in presence of faults.

R3: Future integration by merging: *An FCM can be integrated only with its siblings.*

For example, two *methods* in different *objects* cannot be integrated. This prevents methods not related to an object to be integrated within the same object.

The rationale for **R2** and **R3** follows. As the FCM hierarchy is created to clearly demarcate FCMs and their interactions, if, for example, an *object* FCM has two parents (which in object-oriented terms may imply that there may be two pointer references to an object), then it may suffer from what is called *aliasing* that occurs when an object can be accessed through different names. Hence, boundaries become unclear and it becomes very difficult to prove any properties and the number of possible sites which can potentially corrupt the object FCM increases. What should be clear here is that since *object* is a dynamic concept, it should *not* have two parents or else problems may arise. However, this by no means implies that two tasks cannot share the same class. One possible technique to tackle this problem is the use of the *Clone* method.

A *method* level FCM cannot have two parents since, if another object FCM accesses the method, then there is a possibility of corruption of the data space if the method is *risky* (see Section 4).

Furthermore, sharing of a common code segment by two FCMs of different criticality is not desirable. Therefore, care should be taken when programming to avoid the effects of problems such as aliasing. 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 via its parent and affect higher levels. Due to this, each level represents a different level of abstraction, which simplifies verification and validation of FCMs at all levels. Also, verification and validation of module dependability can be performed independently of other modules at the same level.

R4: *If a method FCM is needed by more than one parent, then a copy of that method is replicated and integrated separately with the other parent(s). For other FCMs (at other levels), a reference is made to the required FCM.*

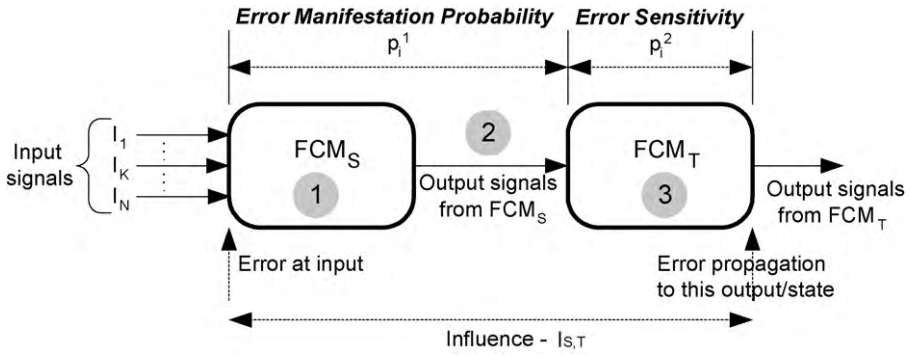

Fig. 3. Error propagating from input of FCM_S to output of FCM_T.

If a method is needed by two different objects, rather than communicating between them (and possibly corrupting the other object data space), a copy of the method FCM is created and integrated with the new parent. On the other hand, a pointer reference is made to an object, by having a new instance declaration in the new parent task.

R5: *Whenever an FCM is modified, its parent FCM, and only its parent, needs to be tested, including the interfaces with its siblings.*

This follows directly from rules **R2** and **R3**, since a change in any FCM affects its interactions with its siblings, but not with children of other parents. Thus, there is a possibility of introducing new errors and transmitting these to its siblings; these interactions need to be tested, to ensure that faults can be contained even after the modification. However, due to the nature of the hierarchical composition, only the parent is affected; because different levels deal with different kinds of fault, not even more remote ancestors must be checked.

With **R5**, we get the very desirable property of fault containment. However, this gives rise to the problem of designing error detection mechanisms (such as EAs) that can detect all harmful errors. In fact, in Jhumka et al. (2002a), we introduced the concept of *globally consistent assertions*, and globally consistent executable assertions can be shown to be complete, i.e., they detect all errors that can potentially harm the system. Thus, adding such detection mechanisms will ensure that errors do not propagate beyond a predetermined boundary.

5.2. Horizontal (Local) integration of FCMs across different SW functions

In the previous section, we have considered issues involved in integrating FCM modules at one level to yield larger FCM modules (at a higher level) within a given SW function i.e., vertical integration. We now consider factors that affect horizontal integration, i.e., consolidating SW FCMs at the same level.

Horizontal integration especially aids in understanding how FCMs, at the same level interact with each other, and assists in determining the impact a fault or error in one FCM has on another FCM by quantifying the influence across FCM's. There are two possibilities of quantitatively assessing the impact of an error in one source FCM (FCM_S) on a target FCM (FCM_T), namely *(i)*FCM_S has a direct influence on FCM_T or *(ii)*FCM_S indirectly influences FCM_T.

For each of the following, we define corresponding metrics that capture the level of interaction involved.[5] At this point, we are mostly interested in the interaction between two arbitrary FCMs at the same level, FCM_S and FCM_T, especially of the impact of a fault in FCM_S on FCM_T.

Definition 1. *Influence* of a source FCM (FCM_S) on a target FCM (FCM_T) is defined as the probability (of error propagation) of FCM_S *a* ffecting FCM_T when no other FCM at that same level is considered. The influence of FCM_S on FCM_T is denoted by $FCM_S \triangleright FCM_T$.

The influence metric quantifies the direct interaction of FCMs in terms of the impact a fault originating in FCM_S has on FCM_T. But two FCMs may interact indirectly via other FCMs. For this scenario we propose the *separation* metric.

Definition 2. *Separation* of FCMs is the probability (of error propagation) of one FCM (FCM_S) *not affecting* another FCM (FCM_T) when *a* ll other FCMs at the same level are considered. Separation between FCM_S and FCM_T is denoted by $FCM_S \vdash FCM_T$.

5.3. Measuring the influence between FCMs

To measure the influence of FCM_S on FCM_T, all factors F by which FCM_S can affect FCM_T (e.g., shared memory, message passing) need to be determined. For each factor F_i, a corresponding probability (p_i) is evaluated. This probability represents the possibility of an error in FCM_S to affect FCM_T via F_i. Generally, the error manifestation process consists of three basic phases, as shown in Fig. 3, namely (1) fault (or error) occurring in FCM_S, (2) error transmission to the input of FCM_T and (3) error transmission resulting in error in FCM_T.

In Jhumka et al. (2001), we have developed an approach that helps in experimentally estimating, using fault injection, the influence values between FCMs. To model the error occurrence (phase *1*) and transmission phases (phases *2* and *3*), two metrics are defined, namely (a) *error manifestation probability*– p_i^1, and (b) *error sensitivity*– p_i^2. Error manifestation probability characterizes the source FCM (FCM_S), and error sensitivity the target FCM (FCM_T), as illustrated in Fig. 3.

In general, a (transient) error occurs at (one of) the inputs of the source module and may propagate via F_i to the inputs of the target module, where an error may occur. The probability of an error to propagate out of F_i, denoted by $p_i^{I_k}$, is as follows:

$$0 \leq p_i^{I_k} = Pr\{i|I_k\} \leq 1 \quad (1)$$

where $p_i^{I_k}$ is the probability of an error in I_k, the k th input of the FCM, to propagate out via F_i.

5.3.1. Error manifestation probability – p_i^1

We define the *error manifestation probability*, p_i^1, of a source FCM (FCM_S – see Fig. 3, phases *1* and *2*) as the probability of an error occurring at the input of FCM_S to propagate, via F_i, to the input set of the FCM_T. This metric is important as it provides pertinent information of how often FCM_S allows errors to propagate. An FCM with high error manifestation probability is a potential candidate for replication or be equipped with EDMs and ERMs. As each input is assumed to have equal likelihood of being erroneous, we can

[5] We use the notation FCM_S for the S th FCM at the current level, i.e., if we are considering influence at the method level, then FCM_S is the S th method of the object.

expand and simplify Eq. (1) to obtain an overall expression for *Error Manifestation Probability*, denoted by p_i^1:

$$p_i^1 = (Pr\{I\}/N) \cdot \sum_{k=1}^{N} Pr\{F_i|I_k\}, \quad (2)$$

where N is the number of inputs of FCM_S, and $Pr\{I\}$ the probability of an error occurring in the input set I of the FCM. In fault injection experiments, $Pr\{I\} = 1$ whereas, if field data or life testing has been performed, $Pr\{I\}$ can be modified accordingly. Also, in case the error probability distribution is known for the inputs, the $(1/N)$ weight for each input is readjusted to reflect the distribution.

5.3.2. Error sensitivity – p_i^2

Once the error has propagated via F_i to the input set of FCM_T, the probability of an error occurring in the state of FCM_T is known as *error sensitivity*, denoted by p_i^2. This metric's importance is in generating information regarding how vulnerable FCM_T is to errors propagating from FCM_S. Target FCMs with high error sensitivity should be protected from errors propagating from source FCMs which is another likely candidate for replication or be equipped with EDMs and ERMs. More details about error propagation in SW is found in Jhumka et al. (2001).

Now that we have introduced estimates of the error manifestation probability p_i^1 and the error sensitivity p_i^2, we can calculate the probability $I_{S,T}^i$ associated with each factor F_i for an error to propagate from FCM_S to FCM_T. This probability is defined as follows:

$$I_{S,T}^i = p_i^1 \cdot p_i^2 \quad (3)$$

Intuitively, this probability gives an indication of how tightly coupled are two directly cooperating FCMs. To increase fault containment, this value should be decreased (possibly below a certain threshold, as governed by the system requirements). Having obtained individual influence probabilities (through different factors), the overall influence $I_{S,T}$ is thus obtained as:

$$FCM_S \triangleright FCM_T = 1 - [(1 - I_{S,T}^1)(1 - I_{S,T}^2)\ldots(1 - I_{S,T}^n)] \Rightarrow I_{S,T}$$
$$= 1 - \prod_{i=1}^{n}(1 - I_{S,T}^i) \quad (4)$$

Note that the value of influence is asymmetric, i.e., $FCM_S \triangleright FCM_T \neq FCM_T \triangleright FCM_S$. For example, one task may use the results provided by another task, but not vice-versa. If FCMs are represented by nodes in a graph, then labeled unidirectional edges represent the influence between them, allowing for graphical representation of error propagation in the system.

At this point, we have divided the influence metric into two sub-metrics which can be estimated on an experimental basis using fault injection. The error transmission probability is estimated using the following procedure: (i) in each input I_k of FCM_S we inject an error (one input at a time, i.e., no multiple errors), (ii) We observe the state and output signals of FCM_S and the state and outputs of FCM_T and use Golden Run Comparison (i.e., comparing an injection run with a *golden* reference run) in order to detect when errors have occurred in either one. Let the number of injection runs where errors in the output of FCM_S and in the state and output of FCM_T have been detected be denoted $n_{err,S}$ and $n_{err,T}$ respectively. The total number of injection runs is denoted n_{inj}. Then, we can estimate the error manifestation probability as $p_i^1 = n_{err,S}/n_{inj}$ and the error sensitivity as $p_i^2 = n_{err,T}/n_{inj}$. We have performed experiments on a real embedded software for an aircraft arrestment system which shows that such an approximation and the analytical model correlate (Jhumka et al., 2001).

Once influence values are determined, the next step is to reduce influence between FCMs so that overall system dependability is enhanced. Since the modules may not have been designed with dependability as main driver, transformations (such as addition of error handling mechanisms) may be needed to convert the modules into real FCMs. In the following sections, we describe potential techniques usable for reducing the influence at various FCM levels. The two metrics, error transmission probability and error transparency, allow identification of vulnerable modules to be protected against propagating errors. Once identified, identification of specific locations for EDMs and ERMs is facilitated.

For the simplicity of the presentation without loss of generality, we neglect the feedback loops of influence in the structure of the SW graph (introduced in the next section). However, our definition can be extended in a straight forward way to cover these cases as well. This problem can be seen as similar to a network flow problem (Ahuja et al., 1993). This problem is represented as a weighted directed graph (WDG) $G = (V, E)$ where the edges E are represented with the capacity of flow and vertices V represent the nodes of the network. It has a source node and a sink node where the overall flow is defined as the net flow entering the sink node.

The analytical model of calculating the influences across modules also resemble the error detection probability in inputs and outputs of combinatorial and sequential circuits using random testing (David, 1998; Ismaeel and Breuer, 1991). According to Ismaeel and Breuer (1991), the probability of detecting an existing input/output circuit fault for m input vectors is $1 - (1 - q)^m$, where q is the probability that a single random input/output vector detects a fault.

5.4. Measuring separation

To this point, we have focused mainly on FCMs directly interacting with each other. However, FCMs may also interact indirectly via other FCMs. To capture this kind of interaction, we introduce the *s* eparation metric, which is complementary to the influence metric.

To measure separation between FCMs at level *i*, a labeled DAG (*s* eparation analysis graph) is created containing one source node and one sink node; nodes represent FCMs at that level, with an edge for each influence pair, from the influencing FCM (source node) to the influenced FCM (target node). Edges are labeled with the influence value. Other nodes in the separation analysis graph represent FCMs through which the source and target FCMs interact. When separation between pairs of FCMs are determined, a *separation graph* can be built to visually represent the separation information available, allowing for vulnerability assessment in the system (Jhumka et al., 2001). The total separation, including transitive contributions, can be calculated as follows:

$$FCM_S \vdash FCM_T = (1 - I_{S,T}) \cdot \Pi_k(1 - I_{S,k}I_{k,T}) \cdot \Pi_{l,m}(1 - I_{S,l}I_{l,m}I_{m,T}), \quad (5)$$

where $I_{S,T}$ denotes the influence between FCM_S and FCM_T. At some point, higher-order terms are likely to be small enough to be negligible. The separation value gives an accurate estimate of the interaction between FCMs, as all FCMs at the same level get considered. Reduction of influence between two FCMs will increase their separation; it is possible to increase separation by reducing influence between other FCMs through which the two interact.

5.4.1. Reducing influence at method level

At the *method* level, the main error transmission mechanism is through data space. Since the scope of the data is confined within the boundaries of the object, there is very limited parameter passing between methods in an object. When parameters are passed (e.g., recursion, or object-valued parameters), one way to reduce

influence of one method on the other is via combined use of redundancy and voting techniques to detect errors in data or using EDMs and ERMs.

5.4.2. Reducing influence at object level

One way an object influences another is through parameter passing. Thus, a possible approach to reduce influence is to have run-time checks on the parameters such as EAs. Another possible problem may be passing of object-valued parameters, necessitating avoidance of aliasing.

5.4.3. Reducing influence at task/process level

Influence factors at *task/process* 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 EDMs and ERMs 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 can be minimized by using preemptive scheduling. Note that most techniques used at the *task* level are also relevant at the *process* level.

6. Systematic allocation of SW modules onto HW nodes

We now describe the systematic allocation approach of assigning different SW modules onto suitable HW nodes. The co-location of HW and SW now specifically entails consideration of SW FCM attributes such as dependability, criticality, and timing specifications, as they relate to the available HW paradigms. In a general sense, the overall problem is one of constrained resource mapping of specified HW and SW elements such that overall dependability requirements are achieved.

6.1. SW and HW graphs

To facilitate the mapping, two graphs are created, one for SW FCMs, and one for the available HW resources, that have been structured using a HW fault containment regions model.

6.1.1. SW graphs

For the SW model, a WDG of process-level FCMs is created, since by assumption, all tasks, objects and methods for a given process are necessarily co-located. Consequently, the *SW graph* consists of a set of processes and their interactions and communications. Inter-process communication is characterized by a WDG, $G = (J, E)$, having the process types as vertices V, and an edge between processes j_s and j_t, if they communicate. At process level the timing properties is represented as (t_i), which is the triple of $t_i(EST_i, CT_i, D_i)$.

EST is the earliest possible time that a process can start its execution. *CT* is defined as the amount of time required by a process to complete the execution on a particular processor. *CT* depends on functional complexity of the process and on the speed of the processor running it. *D* denotes the deadline by which a process or an application must finish the execution. A system designer estimates the values of these properties based on expertise.

In such a graph G $e_{ij} \in E$ is an edge between two process vertices $(v_i, v_j) \in V$, which is the notion of both of influence (I_{ij}) and communication data $(b_{i,j})$ (bytes) between processes. I_{ij} denotes the cumulated conditional probability of error propagation from the source process j_s to the target process j_t, either via message passing or shared resources, assumed that j_s is in a erroneous state. An estimation of determining this values has already been described in the previous section. $b_{i,j}$ is the volume of data required for communication between processes, for instance measured by the maximal total size of information to be transferred per execution cycle. In case a process from the SW graph needs to be replicated for FT requirements, a new vertex is created in the graph for the replica. Replicas are connected by edges of weight 0; there is no edge in the case of non-influence. We assume at this point that protocols ensuring replica consistency (such as atomic broadcast; Cristian et al., 1985, etc.) do not introduce additional influence on a target FCM.

6.1.2. HW graphs

For the HW graph, a set of nodes $N = \{n_1, \ldots, n_k\}$ can be modeled as an interconnection *HW graph* that represents limited HW quantity provided by the processor. The measure of limitation can be in time (e.g., a certain amount of CPU time is assigned) or in space (e.g., a certain memory region is assigned to a partition). The selection of the HW instances depends on parameters like computation, type of CPU, power consumption, failure rate, size and cost. 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,[6] and, if that is possible, then (b) what is a good mapping? Essentially, we are interested in finding a suitable mapping (not necessarily the optimal one but hopefully a near-optimal one) satisfying the attributes of the different SW functions.

6.2. FCM attributes

Each FCM has an associated set of attributes (e.g., criticality, dependability) that need to be combined during integration. Although different attributes are combined differently, the resulting FCM will usually have the most stringent component values (e.g., max criticality, max dependability), 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. The use of FCM attributes while integrating SW and HW is presented below. We first describe the process of utilizing the obtained influence parameters. Next, we propose heuristics demonstrating the viable integration options, and develop approaches to assess the obtained mappings.

6.3. Collocating SW nodes

As the intent is consolidation of SW, we start the process by first conducting integration (vertical [global] and horizontal [local]) within the SW graph to obtain the smallest SW graph in order to map onto the HW. This is the process of clustering the SW nodes into a smaller group. The process of combining multiple SW nodes into clusters to be co-located on a processor involves several considerations, such as influence values between two communicating FCMs, satisfaction of constraints, e.g., timing constraints. Also, two nodes connected by an edge of weight 0 cannot be combined, as the nodes contain replicas of the same SW module, and must be mapped onto distinct HW nodes.

As FCMs are added to the cluster, internal influences disappear (Fig. 4 for FCMs within the dotted area). When processes p_2 through p_6 are combined, their internal influences are no longer visible; however, the influence of the combined processes (those within the cluster) on process p_1 is still significant. If several cluster nodes had individual influences on a common neighbor, those influence values need to be combined; for example, in Fig. 4 influences of processes p_2 and p_4 on process p_1 must be combined. The resulting influence of the cluster C, made up of different processes (denoted by FCM_i), on node/process FCM_t (a target FCM such as p_1 which is

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

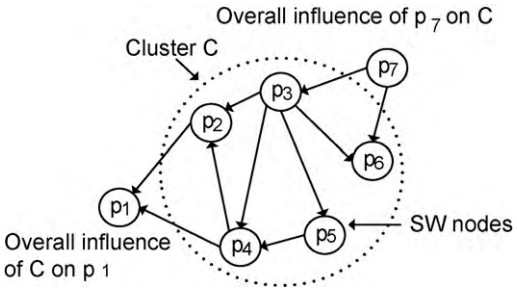

Fig. 4. Combining SW nodes.

shown in the outside of the cluster of Fig. 4) is given by:

$$FCM_C \triangleright FCM_t = 1 - \Pi_i(1 - (FCM_i \triangleright FCM_t)) \Rightarrow I_{C,t} = 1 - \Pi_i(1 - I_{i,t}) \quad (6)$$

First we calculate the value of separation $(1 - (FCM_i \triangleright FCM_t))$ between cluster process FCM_i and the target FCM_t and then the resulting influence of the cluster on the target process is calculated as shown in Eq. (6). This equation can also be used to evaluate the overall influence of a node external to a given cluster (e.g., P_7 on Cluster C). At the end of cluster formation process, there can be multiple clusters (as some FCMs could not be combined to one particular cluster because of timing constraints say, leading to a non-schedulable cluster). These clusters can then be directly allocated onto available HW nodes provided that the HW nodes have the available resources to host the corresponding cluster. This can be checked by verifying if the sum of the memory requirements of all the processes in a cluster is less than or equal to the memory capacity of the HW node. We describe this process in detail using an illustration in Section 7.

6.4. Dependability-driven SW–HW Mapping

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.[7] This helps in making the right choices during the mapping process and also in ascertaining trade-offs. The importance of various criteria may differ, depending on the application under consideration, but these criteria include:

- **Satisfaction of constraints**: this is the satisfaction of hard constraints which implies the absolute constraints on behavior, whether semantic, temporal, or others. 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**: as our main objective is to achieve dependability by design we endeavor to minimize the influences between different modules. The influences among FCMs forming a single cluster is zero. Assigning highly interacting SW FCMs on the same node reduce the fault/error propagation probability across nodes, i.e., the fault/error does not propagate but contained within one node. Low influence values between nodes implies good fault/error containment.
- **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. However, the risk of non-critical tasks affecting a critical one needs to be ascertained and minimized as well, possibly through techniques such as partitioning (Rushby, 1999) and multi-level security (Totel et al., 1998).
- **Resource utilization**: resources should be utilized from different points of view, e.g., utilizing the communication/bandwidth resources, utilizing the number of processors, utilizing load balancing and power. Load balancing is the technique where processes are evenly distributed among the processors to leave as much slack as possible in the mapping. This slack can be utilized for fault-tolerant systems where dynamic checkpointing or re-execution of processes is applied in presence of faults.

The above mentioned criteria also instigate the selection of attributes in case of multi criteria decision procedure analysis. At this point, there can be one or more suitable mappings satisfying some of the above criteria. Thus, the problem of determining the goodness of the possible SW $\leftrightarrow$ HW mapping(s) arises.

6.5. Attribute evaluators

We first quantify the amount of each attribute in each mapping. Here, we assume that we have n evaluators, one for each attribute. Applying the n evaluators to a mapping results in an n-dimensional vector. Note that the quantity of each attribute needs not range from 0 to 1, since it is dependent on the evaluators used. More formally, denoting the ith attribute by a_i, the ith attribute evaluator by E_i, and the quantity of the ith attribute by q_i, then $E_i(a_i) = q_i$. A mapping is then represented as $(q_1, q_2, \ldots, q_n)$.

We subsequently determine a goodness function for the attribute vector, based upon the system requirements. Each mapping (more specifically, its vector representation) is a point in an n-dimensional space. Among all the points defined by the different available mappings, our purpose is to find the highest one (which may not be any of the maxima of the surface). A trade-off is positive only when going upwards along the surface.

Onwards, Section 8 presents a decision procedure that helps in determining the best available mapping as well as its *goodness* value.

6.6. Mapping SW to HW resources – basic approaches

Considering the FT requirements and the need for, say, threefold replication, the initial SW graph is augmented with three identical SW nodes, linked with 0 edge weights expressing a complete isolation between them. Each of these replicated SW nodes can thereafter be treated separately. Since, invariably, the SW graph has a much greater number of nodes than the HW graph, the SW graph must be condensed (cluster formation) 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* (dependability-driven). This particular problem is often NP-hard (cannot be solved in a tractable manner where a solution can be found in polynomial time; Garey and Johnson, 1979). Consequently, heuristic solution techniques are often utilized. Note also that the clustering process needs to be aware of the HW restrictions imposed by the design. Specifically, even if a given cluster is optimal but violates the restriction imposed by the HW, then the cluster will be rejected. In general the heuristics process considers the processes with most important attributes first (Islam et al., 2009) in the mapping process. In this paper we provide some heuristics on how to build the SW cluster.

- **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

[7] A good mapping supposes the creation of a feasible mapping, hopefully a near-optimal one.

Table 1
Processes (and criticalities) in GAP.

Process	System	Criticality
p_1	Radar Warning	Very high
p_2	Display	High
p_3	Navigation	High
p_4	Radar Control	Medium
p_5	Tracking	Medium
p_6	Weapon	Medium
p_7	Built-in Test	Low
p_8	Data Bus	Low

Table 2
Example attributes of SW modules.

Process	Criticality	FT	*EST*	*D*	*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	20	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

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:** 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. Importance is a function of the different attributes that the mapping satisfies.

These heuristics basically determine how the SW nodes will be clustered together. Once a sufficiently concatenated SW graph is obtained, the next step is to determine the mapping satisfying the constraints of the SW nodes with the HW resources. For example, the processes in the cluster must all be schedulable so that their precedence and deadline constraints are met. If this is not possible on any HW resource, the current partition must be rejected. Since we are assuming homogeneous processors, the mapping is straightforward. In other cases, more complex heuristics are needed for the mapping (Suri et al., 1998; Islam et al., 2009). In Islam et al. (2006) we present a heuristic based systematic resource allocation algorithm for the consolidated mapping of safety critical and non-safety critical applications onto distributed computing platform such that their operational delineation is maintained over integration. Using a real application example in the next section we illustrate the mapping process employing the heuristics H1 and H2.

7. Illustrating a real SW–HW mapping

We now consider a real-life representative application example of an aircraft system that combines navigation, radar and partial flight control functions (Locke et al., 1991).[8] The different processes and their criticalities are listed in Table 1. Using the heuristics **H1** and **H2** defined in Section 6.6, and a set of p rocesses from the GAP model (attributes detailed in Table 2), we demonstrate the general techniques. The same set of example *processes* is used across different techniques, to highlight and compare different methods for combining nodes.

[8] We consider the Generic Avionics Platform (GAP), which is a model of an avionics mission computer system.

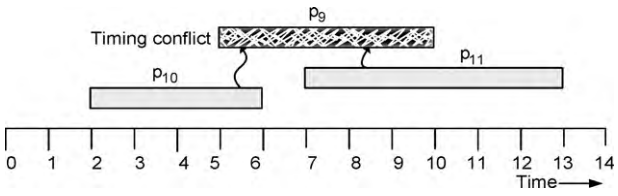

Fig. 5. Schedulability analysis.

We assume a predetermined resource (HW) graph which can be parameterized. However, first we try to perform the mapping considering a fixed set of HW resources. If a feasible mapping is not found with the given resources then we parameterize the HW model by adding new HW nodes. 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 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 the communication costs are too high, then dilation of the mapping may be considered to address performance. Further heuristics can be used to map SW nodes with high communication costs onto (the same or) neighboring HW nodes.

We outline the HW–SW integration and mapping process with an example. Table 2, lists the set of p rocesses $p_1, p_2, \ldots, p_8$ with specific replication (*FT*), criticality and real-time specifications. *Process*p_1 has a high criticality value (*C*), and has to be replicated three times to be run in a TMR (Triple Modular Redundancy) mode ($FT = 3$). *Processes*p_2 and p_3 are of intermediate criticality, with $FT = 2$. The rest of the tasks $p_4, \ldots, p_8$ require no duplication. The other attributes of each process are timing constraints, including *EST*, task completion time *D* and *CT*. The parameters have been chosen to illustrate limits on combining nodes. The timing constraints might also prevent combining specific nodes. For example, Fig. 5 illustrates a situation when two SW nodes (p_{10} and p_{11}) with timing requirements $\langle 2, 10, 4\rangle$ and $\langle 7, 14, 6\rangle$ ($\langle EST, D, CT\rangle$) are already combined to be assigned onto a given processor. If we try to combine another SW node (p_9) of timing requirements $\langle 5, 11, 5\rangle$ with this cluster then they cannot be scheduled on the same processor due to timing constraints violation. Thus, p_9 must be clustered with other nodes.

Note that criticality and replication of the processes (Table 2) do not have to be directly related. If, for instance, a process has substantial state, the replication may result in more risks because of synchronization and communication (and perhaps inhibit rather than promote correct operation and completion). In this case, another method the ensures fault free operation have to be considered instead.

Initially, eight SW nodes are created, one for each *process* of Table 2, linked through edges weighted by influences on other *processes*. For this example, influences have been randomly assigned; in reality, techniques such as those used in Jhumka et al. (2001) would be used to generate real values.

Fig. 6(a) is a graph containing all the processes from Table 2. In Fig. 6(b) the node p_1 is replicated 3 times to satisfy its fault tolerance requirements. Edges with neighbors are also replicated. The three replicas are linked with edges with influence value 0. The total number of nodes of the SW graph in Fig. 6(b) is 12.

Let us assume that the available HW topology to be a strongly connected network with 6 HW nodes. Thus, the 12-node SW graph need to be reduced in size to map onto the 6 HW nodes, using the techniques discussed in Section 6.6; the choice of technique used depends on the most important attribute of the application. When

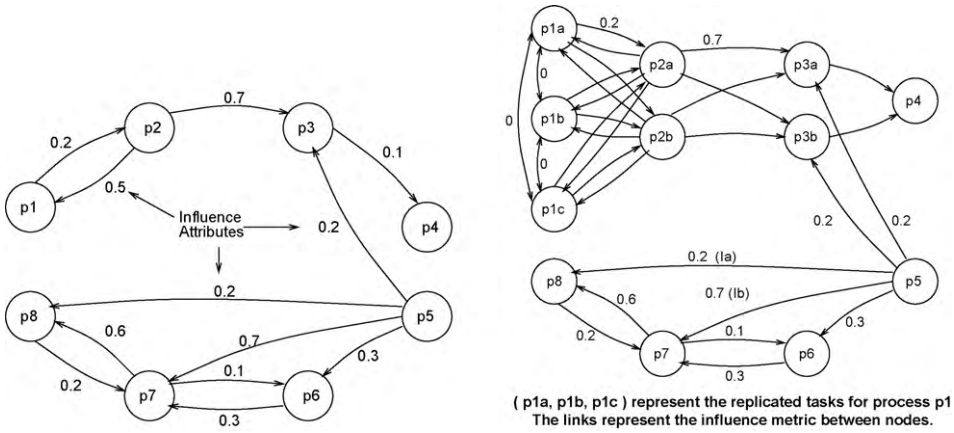

Fig. 6. (a) Initial SW nodes and (b) illustrating influence in SW node linkages.

combining any two nodes, we must nonetheless check the values of all attributes, since certain combinations of nodes may be infeasible. For example, if p_2 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 cannot be combined. Several well-known scheduling algorithms can be used to check the feasibility of scheduling sets of these p rocesses on the same processor (Stankovic et al., 1994). During the HW–SW mapping process, we have presented the constraints satisfaction technique including schedulability analysis in Islam et al. (2006, 2009).

Integration trade-off: While combining SW nodes, some trade-offs might be necessary. For example, it may be preferable to map two critical *processes* onto different HW nodes, but that may however not be possible since both have to be replicated (for FT), and the number of HW nodes is limited. Specifically, if the HW has four 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 the need for a resource present on only one processor (i.e., satisfaction of binding constraints) or a need of a large communication band.

The next section presents three possible approaches of using the heuristics **H1** and **H2** (as defined in Section 6.6) to conduct the HW–SW mapping/integration. Here we are primarily interested in illustrating viable integrations. After developing the basic assessment procedure for a given integration, in Section 9 we will re-visit these three possible integrations to assess the goodness of the integration achieved by these three approaches.

7.1. Combining Nodes using Heuristic H1: Approach A

As the provision of dependability is a primary concern, the criteria for fault containment predominate (Chillarege et al., 1995; Randell, 1975). Thus, combining nodes with high mutual influence values (sum of influences in each direction) reduces the probability of errors being transmitted across HW nodes, creating *fault containment regions* in HW. The graph in Fig. 6(b) can be directly reduced based on influence values.

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. 7. The new influence attributes for the combined processes are obtained through iterative use of Eq. (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. 7 and 8 show successive stages of this process. Note that the *processes* with 0 influence [(p_{1a}, p_{1b}, p_{1c}), (p_{2a}, p_{2b}), and (p_{3a}, p_{3b})] get mapped to distinct HW nodes. These processes have been replicated according to their level of criticality. Fig. 8 shows a six-node HW graph after several stages of SW node combinations. The resulting mapped nodes in the graph satisfy the objectives. Depending on the size of the HW graph, the SW graph can be further reduced; this however raises the issue of trade-offs in integrating SW beyond a HW resource threshold.

7.2. Scheduling Critical Processes on Separate Nodes: Approaches B and C

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. Also, minimizing the number of critical *processes* scheduled on one given processor also minimizes the number of *c* ritical processes lost due to a crash fault of that processor. These critical *processes* can also be allocated separate portions of memory to avoid faults due to memory cell malfunction. This guides the process by which the graph in Fig. 6(b) can be integrated into six nodes with total criticality on each HW node reduced as much as possible. The processes are considered and ordered according to the importance of FCM attribute in this case according to the criticality. *Approach B*

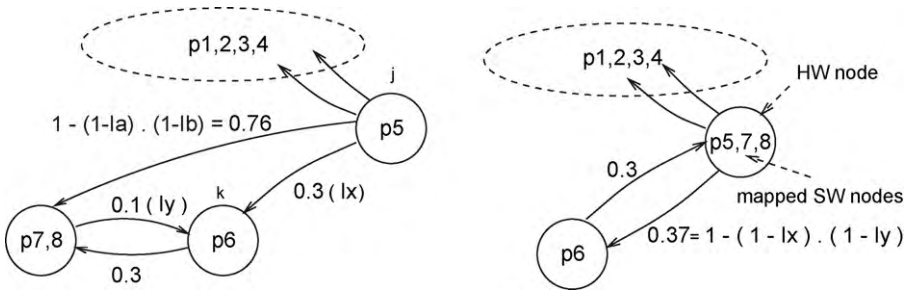

Fig. 7. Using influence to combine the SW nodes to match the HW resources.

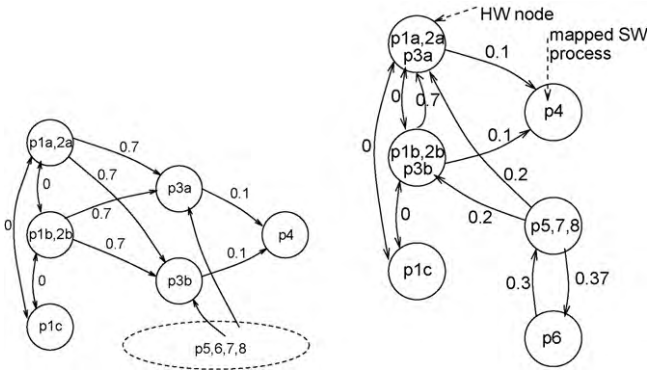

Fig. 8. Reducing SW graph to match HW resources: Mapping A.

using heuristic **H2** is thus as follows:

B1 List *processes* in descending order of criticality.
B2 Combine most critical *process* with least critical *process*, the second most critical *process* with the second least critical *process*, and so on.
B3 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.
B4 If a high criticality $processp_h$ cannot be combined with a low criticality $processp_l$ due to conflicts (e.g., timing constraints), then combine p_h with the *process* preceding p_l in the criticality list.
B5 In the next stage, the sets of *processes* can be ordered based on a summary criticality (e.g., the highest criticality, or the sum). The previous steps can then be repeated until a desired number of nodes is obtained.

In this 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 replicated and cannot be combined, thus leading to a conflict (due to FT constraint). To resolve this, the next higher criticality $processp_{2b}$ is combined with p_{3b} with p_{3a} is combined with p_4. The resulting graph is shown in Fig. 9(a).

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 or refine the mapping. One such technique, *Approach C*, is:

C1 Compute an ordered list of SW nodes.
C2 Place the nodes which should preferably be mapped onto the same node adjacent to each other.
C3 Next, map SW nodes onto a HW node starting at the top of the list maintaining their compliance to the specified constraints.

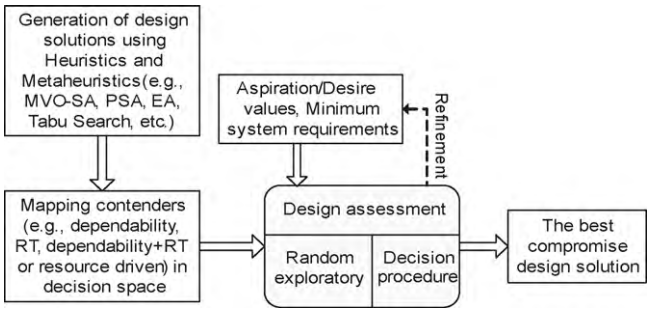

Fig. 10. Generic assessment framework.

For example, the graph in Fig. 6(b) can be straightforwardly reduced to Fig. 9(b) if only the timing attributes are considered.

8. Quantification and assessment of a mapping's goodness

We have described the basic framework for developing the SW graph and conducting the SW/HW mappings based on the proposed heuristics. Once the different possible mappings are available, as explained in Section 6.4, the system designer needs to choose the best mapping (satisfies the minimum system requirements) among the contending ones. In this paper we have described heuristics for creating the mappings. However, without loss of generality mappings can also be generated by using other meta-heuristics and algorithms, e.g., Multi Variable Optimization (MVO) approach (Islam and Suri, 2007), PSA (Pareot Simulated Annealing) (Czyzak and Jaszkiewicz, 1998), Evolutionary algorithms (Zitzler et al., 2003; Jhumka et al., 2005), Tabu search (Izosimov et al., 2005). The resulted contending mappings can be dependability driven, RT driven, dependability and RT driven, etc. Thus, there is a need to develop two specific aspects, namely:

(1) Means to quantify the *goodness/QoS* of the achieved mapping, and
(2) Ability to assess the relative suitability of each heuristic (and any other subsequent heuristics), given a certain specific application and/or criteria.

The generic framework for assessing the goodness of mappings is depicted in Fig. 10. The overall objective is to be able to identify and ascertain the trade-offs involved over each mapping strategy.

In the subsequent sections, we utilize a decision theoretic approach to develop two new techniques to assess the mappings. The first method, termed as the *Random Exploratory Technique* addresses the *goodness* of the mapping in an intuitive manner, analogous to a search procedure. We then present a step-by-step decision procedure that provides a formal framework for quantitatively assessing the goodness of the mappings. The interactive

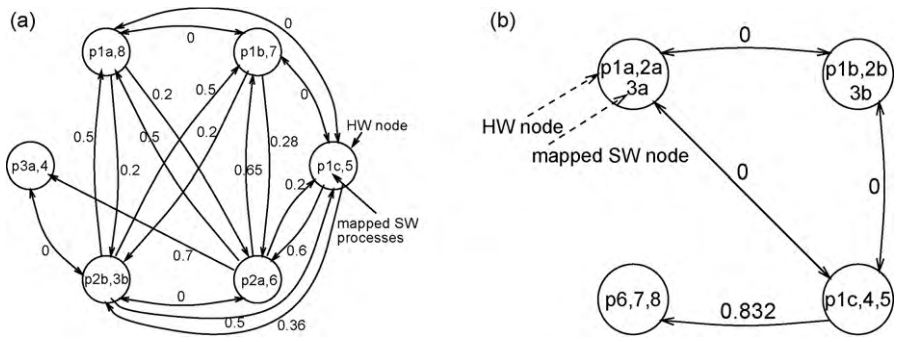

Fig. 9. (a) Criticality-oriented integration: Mapping B and (b) Mapping C.

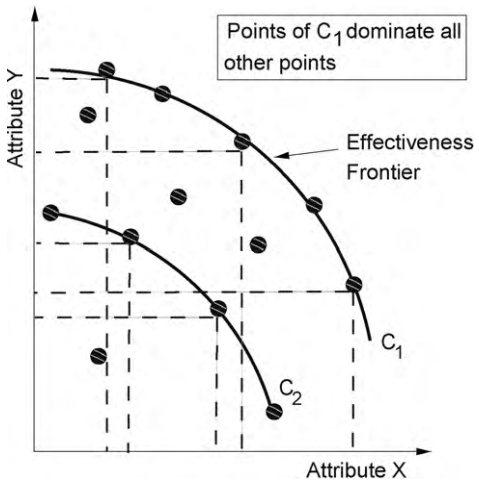

Fig. 11. Dominance and effectiveness frontier.

decision procedure presented in Section 8.3.1 is an application of decision theoretic principles. Overall, the decision procedure takes a set of possible mappings, and tries to evaluate their relative merits, with regards to the relevant attributes. The decision procedure is a heuristic that allows quantification of the mappings, i.e., evaluating the goodness of each mapping, whereby the best compromised mapping among the contenders is chosen. Given that we are not looking for an optimal mapping (but hopefully a near-optimal one), a heuristic or a metaheuristic that induces a partial ordering over the set of mappings is adequate. The novelty with the decision procedure is that it helps the system designer in structuring the attributes of interest as per their importance. For reason of space, the definitions are omitted, but they can be found in Keeney and Raiffa (1993).

For the clarity of discussion, we only present two terms, namely (i) *dominance*, and (ii) *effectiveness frontier*. A vector V_1 *dominates* a vector V_2 iff there exists one element of V_1 which is strictly greater than the corresponding element of V_2, with all other elements of V_1 being greater than or equal to each corresponding element of V_2. The set of dominating vectors is called the *effectiveness frontier*. These are depicted in Fig. 11. The points of vector V_{C1} on effectiveness frontier C_1, which also known as *Pareto optimal set* dominates all other points in the graph as well as dominates the points of vector V_{C2} on curve C_2.

8.1. Decision theory: overview and relevance

Decision theory provides a framework for reasoning about preferences and is based on the axioms of probability and utility. Probability theory provides a framework for coherent assignment of beliefs with incomplete information whereas utility theory introduces a set of principles or simple axioms/rules for consistency among preferences and decisions (Horvitz et al., 1988).

In this study, we are primarily concerned with outcomes in the presence of certainty. Decision under uncertainty means that choice of a given alternative does not guarantee a known outcome, but rather entails selection of a given probability distribution of outcomes.

In the presence of certainty, certain properties hold. First, there is the axiom of *orderability*, which asserts that all pairs of alternatives are comparable, even when described by vectors of attributes. Thus, given two alternatives A_i and A_j, the decision maker either prefers A_i to A_j ($A_i \succ A_j$), or prefers $A_j \succ A_i$, or is indifferent ($A_i \approx A_j$). Second, there is the axiom of *transitivity*, assuring that if $A_i \succ A_j$, and $A_j \succ A_k$, then $A_i \succ A_k$. Similar conditions happen when one or both of the relationships are indifference.

In the presence of certainty, we then have a weak preference ordering, thus the existence of a scalar value function, G, mapping all possible alternative attribute vectors V_i to scalar values $G(V_i)$, such that the decision maker will always prefer the outcome with the highest scalar value (depends on whether the attributes are positively (highest value) or negatively (lowest value) oriented). This value function is sometimes known as the *worth* or *u* tility function; in this paper, it is termed as the *goodness* function G. Thus, we need to determine a *goodness* function G that can compute the goodness value of a mapping. Intuitively, the goodness function will be system-dependent, since different systems may warrant different trade-offs. The trade-off study cannot be dissociated from determination of the goodness function, since the goodness of a mapping will depend on the *quality* of the trade-offs made. However, the influence values that underpin our dependability-driven software integration approach are certain. Thus, no uncertainty is involved when computing all possible mapping alternatives.

In the presence of uncertainty, however, neither orderability nor transitivity need hold. Even with certainty, the computation of a precise goodness function may still be infeasible. Thus, we present new heuristic approaches below. There are two possible ways for resolving trade-offs:

- The system designer informally weighs the trade-offs, or
- The designer formalizes the goodness structure and uses this to evaluate the contending alternatives (if any) to determine the best alternative.

We present strategies to handle each option. For the informal approach, we present a technique, termed *random exploratory technique*, that qualitatively determines a mapping as good. The result is an informal approximation to the goodness of the mapping.

The second approach is a systematic structure for determining a (heuristic) goodness function through a 9-step decision procedure. A salient feature of the procedure is that it is iterative, so that a goodness function that is as precise as possible can be obtained.

8.2. Random exploratory technique

This case works well if there are a limited number of contending mappings to evaluate. We assume that the mappings are already in (attribute) vector form. As an example, consider a hypothetical vector V_1 (on a scale of 0–1) that contains these attribute values for the following attributes: (i) fault containment[9] (FC) = 0.5, (ii) criticality (CR) = 0.3, (iii) load balancing (LB) = 0.4, and (iv) slack (S) = 0.2. Hence, the 4-attribute vector V_1 is (0.5, 0.3, 0.4, 0.2). Then, from system specifications, the system designer generates an aspiration vector (V_a) that is indicative of the system's requirements.[10] Let $V_a = (q_1, q_2, q_3, q_4)$, where $q_1 \ldots q_4$ are aspiration values. Using the aspiration vector, mappings dominated by V_a are removed since they do not satisfy system's requirements. From the remaining vectors, the subset of dominated mappings is also removed.

At this point, there are one or more contending mappings. From the system requirements, for a particular vector V_i the system designer informally decides on the attributes for which a trade-off can be made in V_i, such that there is no preference ordering between V_a and V_i. More specifically, let $V_a = (q_1, q_2, q_3, q_4)$ and $V_i = (q_5, q_6, q_7, q_8)$. Assume also that $q_1 = q_5$, $q_2 = q_6$, $q_3 < q_7$ and $q_4 > q_8$. The above problem is then summarized as follows: is trading-off some of q_4 for more of q_3 in V_a ok, such that V_i is obtained? If *t* rue, then mapping V_i is *good* since it is *not dominated*

[9] This is a function of influence.
[10] The system designer may indicate the minimal requirements of the system.

and is *as preferable* as V_a. Otherwise, V_i is rejected as a bad mapping. Another mapping V_j is then considered and the same process performed.

Another possibility is for the system designer to manually generate a new mapping by adjusting the current one.[11] From the resulting mapping, the system designer proceeds with the same decision process as above. Usually, this will be a subjective judgment, allowing trade-offs to be conducted. Note that this involves determining a vector in the effectiveness frontier by randomly probing the frontier – hence its name, random exploratory technique – rather than determining any sort of optimum value. Also, given required system specifications, the trade-offs (even if highly informal and applicable only in the present case) can lead to preference among elements of the effectiveness frontier, which can be included explicitly during goodness assessment.

This technique of determining the goodness of a mapping is not intended to be accurate, but rather to provide a yes/no answer as per the goodness of mappings. There are some important characteristics about this technique. It can be used as a stand-alone technique to ascertain the goodness of the mapping. It can also be used in conjunction with the more formal approach explained in the next section, as a preprocessing computation to prune the set of alternatives.

Next, we present a formal approach to determining the goodness of the mapping by defining a formal framework to derive a goodness function.

8.3. Systematic decision procedure

This is a heuristic process which introduces a *step-by-step* decision procedure and systematically guides the system designer in selecting the best available mapping. We first describe the formation of a utility or value function (in our case goodness function) and then in the next Section 8.3.2 we depict the 9-step decision procedure based on the multi criteria decision theoretic analysis.

8.3.1. A formal structure for the goodness function

In this section, we are concerned about structuring the preferences to simplify the trade-off analysis. As we are concerned about multi attribute optimization, we provide a brief overview of the theory underpinning generation of multi-attribute utility functions. We refer the readers to Keeney and Raiffa (1993) for details about 2- and 3-attributes optimization. Very often, real-world cases deal with more than three attributes. We assume that we have n evaluators, $E_1, E_2 \ldots E_n$, evaluating attributes $a_1, a_2 \ldots a_n$ respectively, such that $(E_1(a_1), E_2(a_2), \ldots E_n(a_n))$=$(q_1, q_2, \ldots q_n)$.

A *goodness function*, G, may be expressed in additive form

$$G(q_1, q_2, \ldots, q_n) = \sum_{i=1}^{n} \lambda_i \times G_i(q_i) \tag{7}$$

where G_i s are single-attribute goodness functions (Keeney and Raiffa, 1993), and $\sum_i^n \lambda_i = 1$ *iff* the attributes are mutually preferentially independent, i.e., trade-off between pairs of attributes is independent of the values of other attributes. The higher the value of λ_i, the higher the importance of the corresponding attribute, i.e., a_i. Thus, to determine a goodness function, each λ_i needs to be determined, subject to the constraint $\sum_i^n \lambda_i = 1$. Also, each function G_i needs to be determined by the system designer. Determining accurate G_i is not an easy process. However, we present some guidelines about how to generate these functions in Section 9.6.3.

[11] This entails reassigning some processes to other processors to emphasize other attributes.

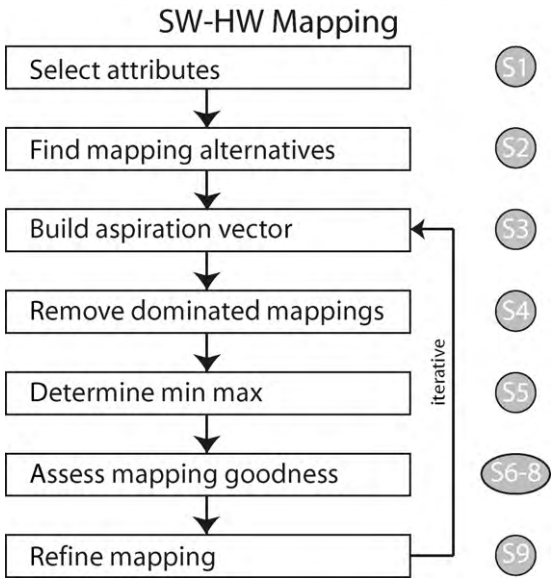

Fig. 12. Our 9-steps approach to finding the best mapping.

8.3.2. An interactive decision procedure for determining goodness

In this section we present a decision procedure that helps choosing the best mapping from a set of contenders. Thus, to construct a goodness function, a number of steps needs to be followed. Fig. 12 depicts these steps in a visual fashion. The different steps are highlighted, together with the iterative process of selecting the best possible mapping.

During this process the knowledge of the user on the system is utilized hence an interactive procedure. The 9-step procedure presented below is a general approach for determining an (approximate) goodness function to ease trade-offs, and is detailed below. We will first state the step operations as depicted in Fig. 12, and then explain any connotations/assumptions that may apply.

S1 Determine all the attributes to be considered. For example, attributes can be fault containment (i.e., (mutual) influence), criticality, load balancing, and so on. Arrange them in order of decreasing importance. Determine, for each of them, whether they are positively-oriented (higher preference for higher score) or negatively-oriented.

For simplicity, we note the following assumptions (1) in the balance of the process description: all attributes are positively-oriented, (2) attributes can be ordered in importance, independent of values, subject to the *requirements* of the system (3) all attributes, taken atomically, have an optimal value of 0 or 1, i.e., are uniformly positive or negative.

S2 Run the heuristics, H_i, to obtain a set of possible mapping alternatives. Represent each of them as an n-attribute vector. If more alternatives are needed, a *search tree* representing the allocation space can be generated, subject to some constraints such as system and temporal constraints. The cost associated with generating all possible mapping alternatives is linear with the number of alternatives to be considered. Also, since we are interested in near-optimal allocation, i.e., a good allocation, exponential complexity for generating such mappings can be avoided. Overall, the cost is not very high.

S3 Determine a vector of attributes that best represents the system's requirements. This vector contains aspiration values and can be seen as a *constraint vector*. The assumption here is the availability of evaluators, E_i. This helps in determining if this mapping is dependability- or criticality-driven, etc., depending on the assignments of values to each attribute. Note that this

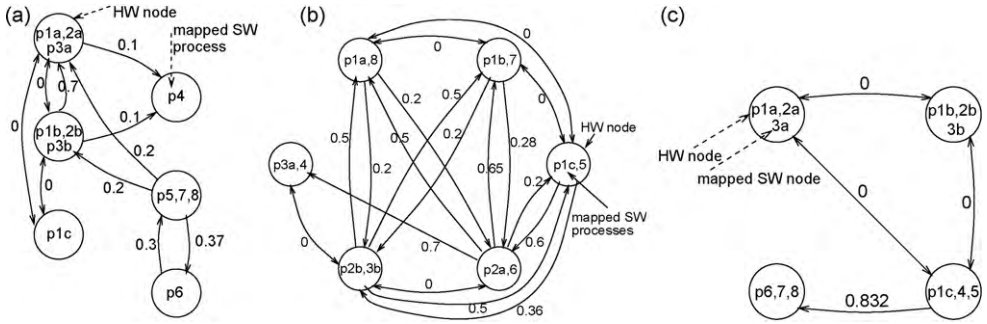

Fig. 13. (a) Dependability-driven: Mapping A, (b) criticality-driven: Mapping B and (c) Mapping C.

vector is doing double duty – both a constraint vector, and a measure of level of importance.

S4 Remove all vectors that are dominated by the aspiration vector, since they do not satisfy the system requirements. From the remaining vectors, remove all those that are dominated, since these do not lie on the effectiveness frontier (i.e., cannot be the best alternative). This aids reducing the number of alternatives to be considered.

S5 For each attribute, determine the minimum and maximum values from the remaining alternatives. This is done to help in determining normalized single-attribute functions (range from 0 to 1). There is a minor assumption here: namely, that, given a (min, *max*) range, the set of values actually encountered will form a reasonably large subinterval of these. In some cases the range of values is $[0, n]$, but with very high probability the value will be very close to, for example, $sqrt(n)$.

S6 Determine the individual goodness value function and the overall goodness function. We need to check for consistency of the goodness function with respect to the aspiration vector provided in S4. The preference structure should be respected.

S7 For each mapping in the set of remaining contenders, insert the attribute values in the goodness function to obtain their respective goodness value.

S8 Select the alternative with the highest goodness value.

S9 If the mapping offers values which can be traded off, then modify the values in S3. Re-execute S4 through S9. Again, there is clear double duty in the aspiration vector. Note that in S6, when determining the goodness function, the λ_i's provide a first round of trade-off. However, since determining a precise goodness function is intractable, S9 allows a further iteration through the procedure such that a goodness function as precise as possible is obtained.

Once a goodness function has been determined, we plug in different mapping vectors to obtain the best available solution. At this stage, we re-use the example of Section 7 to illustrate the complete 9-step decision procedure.

9. Practical utility of the decision procedure: a real example

In order to consistently relate the decision process to the earlier developed heuristics of Section 8.3, we re-utilize the 8-process example presented in Table 2 of Section 7 to illustrate the applicability of our decision procedure presented in Section 8.3.1 to determine the *goodness* of mapping.

9.1. Step 1: selecting attributes

Since our aim is to assess the design of dependable real-time embedded system, the attributes are considered from those perspectives. For embedded systems, there are generally multiple different attributes (dependability, power, space, etc.) to be considered. We consider the following attributes, in decreasing order of importance.

Fault containment (a_{fc}) refers to how well errors are contained within the system. Low influence values imply good fault containment between modules. As our prime driver is dependability, the importance of fault containment is uppermost. From the concepts of error permeability introduced in (Hiller et al., 2004), a relative (or normalized) measure of the error containment capability of a software module can be obtained.

Criticality (a_{cr}) indicates how critical tasks are allocated. High criticality values imply that high criticality processes are not located on the same HW resource.

Slack (a_{sl}) represents how unloaded any processor is. This aspect of the mapping is considered since having some *s* lack in the system for later upgrades is desirable.

Load balancing (a_{lb}) denotes statistical variance of loads on various processors in the system. This attribute, here, is of lesser importance than fault containment and criticality, but nevertheless is considered as a performance aspect.

Communication (a_{co}) represents the communication volume between processors. This attribute is considered from a performance point of view and the inverse of the communication volume is therefore considered, so that high communication attribute values represent low communication volumes.

Note that all the attributes here are taken to be positively oriented, i.e., the higher the value of an attribute, the greater *quantity* there is of this attribute. Step 1 is now complete as we now have a sorted attribute list.

9.2. Step 2: mapping alternatives

Fig. 13(a) shows the resulting mapping after the SW graph in Fig. 6(b) has been condensed to match the number of processors using heuristic H1. On the other hand, Fig. 13(b) shows the state of the processors after running the criticality-driven heuristic H2 on Fig. 6(b) (resulting in the same graph as in Section 7.2). Fig. 13(c) shows the mapping where dependability and criticality were not the most important attributes (as in Section 7.2). Overall, Figs. 13(a)–(c) shows the different contending alternatives and are denoted as *Mapping A, Mapping B* and *M* apping C respectively. To complete Step 2, we summarize the three contenders as attribute vectors.

We assume the existence of relevant attribute evaluators, since although obtaining them is a straight forward process, it depends on the system designer's preference. For example, consider the attribute fault containment: a possible attribute evaluator may be to take the inverse of the minimum influence value between any two nodes (i.e., resulting in maximum fault containment).

Table 3
Different mapping alternatives.

Mapping contenders	Design assessment criteria				
	Fault containment (q_{fc})	Criticality (q_{cr})	Slack (q_{sl})	Load balancing (q_{lb})	Communication (q_{co})
Mapping A	0.75	0.10	0.10	0.15	0.30
Mapping B	0.15	0.80	0.18	0.75	0.10
Mapping C	0.10	0.15	0.30	0.25	0.50

Another example, considering the communication attribute: one possible attribute evaluator for communication may be the inverse of the number of messages or the inverse of the size of sent and received messages per process execution time. Each mapping properties is shown in Table 3 and summarized as follows: $V_X = (q_{fc}, q_{cr}, q_{sl}, q_{lb}, q_{co})$. Let $V_A = (0.75, 0.1, 0.1, 0.15, 0.3)$, $V_B = (0.15, 0.8, 0.18, 0.75, 0.1)$ and $V_C = (0.1, 0.15, 0.3, 0.25, 0.5)$.

Step 2 is complete. The problem now is, given these three mappings, which one should the system designer single out as the best one? Fig. 14 shows the performance profiles of the three mappings. The figure visually highlights the relative importance of each attribute (on a scale of 0–1; any other scale will work as well) in the three situations.

9.3. Step 3: aspiration vector

In this step, the designer needs to indicate the minimal requirements of the system. From system specifications, information pertaining to which processes should be kept separated, etc. are provided. Using this information and the attribute evaluators, the system designer approximates the system requirements in the *aspiration vector*. Assume that the values for the desired preference are $V_a = (0.6, 0.25, 0.15, 0.1, 0.05)$. This vector only indicates an initial level of preference that may be refined, such that this vector reflects a more accurate set of requirements of the system. This process of refining the preference levels is called *preference refinement*.

9.4. Step 4: eliminating dominated alternatives

Using this aspiration vector from Step 3, we remove all mappings which are dominated. In our example, all of them will remain since none of them is dominated by the aspiration vector, and also none of the different alternatives is dominated by another one. Let us consider another *Mapping D* represented by the vector $V_D = (0.4, 0.20, 0.10, 0.10, 0.03)$. This mapping is dominated by the aspiration vector $V_a = (0.6, 0.25, 0.15, 0.10, 0.05)$ in all variables and therefore, *Mapping D* is not a contender for the best and is removed from the list.

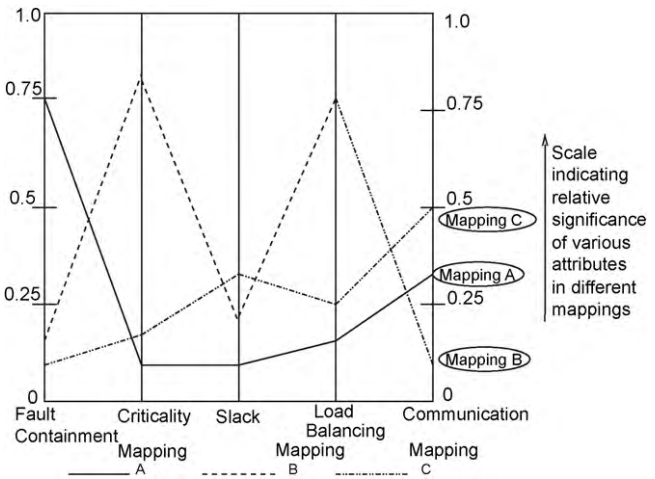

Fig. 14. Performance profiles for the three mappings.

Table 4
Minimum and maximum values for each attribute

Attributes	Minimum	Maximum
Fault containment	0.10	0.75
Criticality	0.10	0.80
Slack	0.10	0.30
Load balancing	0.15	0.75
Communication	0.10	0.50

9.5. Step 5: determining min max

Table 4 summarizes the required information. At this stage, the list of contenders contain only those lying on the effectiveness frontier.

The next step is the crucial one in the decision procedure, as it derives the goodness function that encapsulates the preference (requirements) structure.

9.6. Step 6: assessing the individual goodness function

There are several techniques/methods one can apply to assess or estimate the individual goodness function G_i. In this work, we apply the midvalue splitting technique (Keeney and Raiffa, 1993) in order to determine the value of the attribute's goodness function, which is a comparatively easily applicable method and appropriate for our considered system model. The idea here is to find the s ubjective middle point of different attribute values. We now outline salient sub-steps for generating the generalized goodness function.

9.6.1. Testing for mutual preferential independence

Testing for mutual preferential independence means to determine whether a trade-off between any pair of attributes is independent from the value of any other attributes. If q_1, q_2, and q_3 are three attributes, then trade-offs between q_1, q_2 are independent from the attribute q_3, similarly trade-off between q_3, q_2 should be independent from the value of q_1. To better understand this concept, consider the following example: Assume there are n nodes in the system, with node N_1 having a high computation load, and all other nodes $N_2 \ldots N_n$ have low computation loads. Also, the processes running on N_1 are such that they have a high degree of interaction, such that the mutual influence value between them is high. Any possible trade-off would entail reassigning some of the processes on N_1 to other nodes, thereby balancing the load in the system at the expense of having decreased fault-containment, since highly interacting processes will then be located on different nodes. This is irrespective of whether another node N_i has two high criticality processes running on it, i.e., high criticality value of the system. In this sense, the trade-off between fault-containment and load balancing is irrespective of the criticality value. Hence, these attributes are mutually preferentially independent. Observe that we do not imply that the attributes are independent, however, intuitively, the mutual preferential independence implies that, for a given trade-off, instead of looking at all the attributes at the same

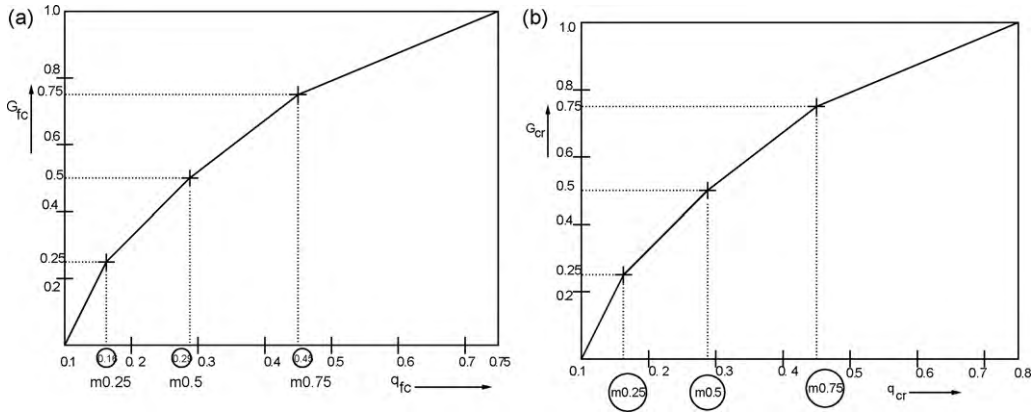

Fig. 15. Graphs of (a) G_{fc} vs. q_{fc} (b) G_{cr} vs. q_{cr}.

time, the system designer can focus only on the attributes of interest. Usually, a small amount of dependency can be ignored within a sufficiently local region of the comparison space (Prasad and McDermid, 1999). However, if there is any dependency between attributes it can be balanced in the overall goodness function where individual functions are added weighing the trade-offs. In the goodness function, we try to adjust some dependency between attributes in order to make the decision procedure applicable. Due to this the goodness functions of the attributes are additive and the presented step-by-step method can be applied to find the best mapping.

9.6.2. Testing for additivity

If the mutual preferential independence condition is satisfied, then this implies that the preference structure is additive (Keeney and Raiffa, 1993), i.e., the goodness function, G, from Eq. (7), can be expressed as follows:

$$G(q_{fc}, q_{cr}, q_{sl}, q_{lb}, q_{co}) = \lambda_{fc} \cdot G_{fc}(q_{fc}) + \lambda_{cr} \cdot G_{cr}(q_{cr}) + \lambda_{sl} \cdot G_{sl}(q_{sl}) + \lambda_{lb} \cdot G_{lb}(q_{lb}) + \lambda_{co} \cdot G_{co}(q_{co}) \quad (8)$$

- $G_{fc}, \ldots, G_{co}$ are single attribute value functions,
- G_i (worst q_i) = 0 and G_i (best q_i)=1, $i \in \{fc, cr, sl, lb, co\}$
- $0 < \lambda_i < 1, i \in \{fc, cr, sl, lb, co\}$ and $\sum_i \lambda_i = 1$

Note that if the mutual preferential independence condition does not hold, other value functions may be more appropriate for modeling the dependent attributes, such as polynomial or probabilistic value functions (Luce et al., 1990). However, these are mostly of theoretical importance and have hardly been used in practice since the axioms in their representation theorems are much more complicated.

9.6.3. Determining goodness function G_i

We now sketch how to decide on function G_{fc} by using the midvalue splitting technique. The procedure equally applies for deriving any other goodness function.

- First, determine the range over which *fc* is defined – Table 4, and we obtain the maximum $q_{fc} = 0.75$ (mapping A) and minimum $q_{fc} = 0.10$ (Mapping C).
- Second, we normalize G_{fc} by letting $G_{fc}(0.10) = 0$ and $G_{fc}(0.75) = 1$.

Next, we want to find the subjective middle point that we will denote $m_{0.5}$. The property of $m_{0.5}$ is such that $(0.10, m_{0.5})$ and $(m_{0.5}, 0.75)$ are differentially value-equivalent, i.e., we seek this value $m_{0.5}$ such that we are willing to pay the same amount to go from 0.10 to $m_{0.5}$ and from $m_{0.5}$ to 0.75. In this case, assume that the value of $m_{0.5} = 0.29$. This midvalue definition is iterated over different ranges. Repeating the process for the range $[0.10, m_{0.5}]$ results in $m_{0.25}$. The same is done over the range $[m_{0.5}, 0.75]$ for obtaining $m_{0.75}$. If more accuracy is needed, we continue on finding the mid value point of different intervals. Also, once we have determined $m_{0.25}$, $m_{0.5}$ and $m_{0.75}$, we can always find the subjective midpoint of the range $[m_{0.25}, m_{0.75}]$ to determine whether $m_{0.5}$ is accurate enough so as to filter out any inconsistencies in the values chosen. Let $m_{0.25} = 0.16$ and $m_{0.75} = 0.45$.

Being a subjectively defined value, we are not aware of any algorithm for computing the subjective middle point of a given interval. Informally, considering the above example, maximum $q_{fc} = 0.75$, and minimum $q_{fc} = 0.10$, which in effect means that the best the designer can get for fault-containment is 0.75, and the worst 0.10. The subjective middle point, informally, represents the cut-off point where the designer is half-satisfied.

The graph in Fig. 15(a) illustrates the function G_{fc} (q_{fc}) for the computed values of $m_{0.25}$, $m_{0.5}$ and $m_{0.75}$. The same process is repeated for each of the different single attribute value function. A similar procedure is repeated comparing values for G_{cr} vs. q_{cr}, G_{lb} vs. q_{lb}, G_{sl} vs. q_{sl} and G_{co} vs. q_{co} – as shown in Figs. 15(b)–16(c).

9.6.4. Determining trade-off factors λ_i

There are a variety of techniques that can now be used to determine the various λ_i's. We first present some notations. Let w_i and b_i be the worst and best values of the ith attribute. Thus, we have $w_i \leq q_i \leq b_i$.

The sorted list from Step 1 provides the relative ordering among the λ_i's as: $\lambda_{fc} \geq \lambda_{cr} \geq \lambda_{sl} \geq \lambda_{lb} \geq \lambda_{co}$.

However we still need to obtain more refined (in)equalities among the attributes to determine their respective values. First, using the preference given in *S*4, this establishes the relations across the attributes. More formally, it is done as follows: Compare these two different profiles, $P_1 = (q_{fc}, w_{cr}, w_{sl}, w_{lb}, w_{co})$ and $P_2 = (w_{fc}, b_{cr}, w_{sl}, w_{lb}, w_{co})$, where w_j identifies the worst value in attribute j, and b_j denotes the best value in attribute j (*S*5). Now, we start varying the value of q_{fc} such that the indifference condition results, i.e., the system designer is indifferent to P_1 and P_2 (they are on the same indifference curve; Keeney and Raiffa, 1993). Suppose the indifference conditions occurs at $q_{fc} = 0.30$. Thus, we can deduce that $\lambda_{fc} \cdot G_{fc}(0.30) = \lambda_{cr}$, since $G_i(w_i) = 0$ and $G_i(b_i) = 1$, for any attribute i. Specifically, $G(P_1) = \sum_{i \in \{fc,cr,sl,lb,co\}} \lambda_i * G_i(q_i) = \lambda_{fc} * G_{fc}(0.30), G_i(w_i) = 0$, for $i \in \{fc, sl, lb, co\}$. For profile P_2, we obtain $G(P_2) = \sum_{i \in \{fc,cr,sl,lb,co\}} \lambda_i * G_i(q_i) = \lambda_{cr} * G_{cr}(b_{cr}) = \lambda_{cr}$, since $G_{cr}(b_{cr}) = 1$. Since the function G_{fc} has already been deter-

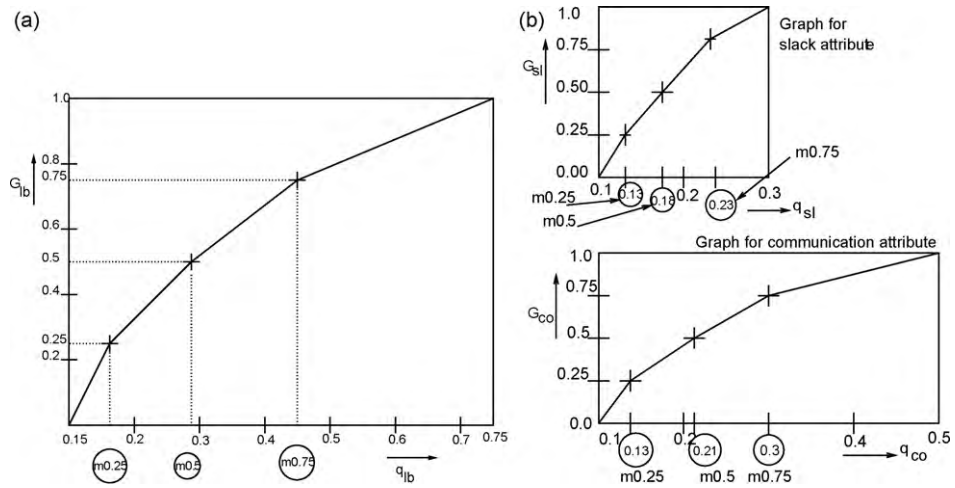

Fig. 16. (a) Graph of G_{lb} vs. q_{lb}, (b) G_{sl} vs. q_{sl} and (c)G_{co} vs. q_{co}.

mined, we can evaluate $G_{fc}(0.30)$ from Fig. 15(a), which is equal to 0.51. Since P_1 and P_2 are on the same indifference curve, we get an equation involving λ_{fc} and λ_{cr}, i.e., $\lambda_{fc} * G_{fc}(0.30) = \lambda_{cr} \Rightarrow 0.51 * \lambda_{fc} = \lambda_{cr}$. The same process is repeated, until the series of equations are obtained. We now determine the proportional relationships between λ_{sl} and λ_{fc}; between λ_{lb} and λ_{fc}; and between λ_{co} and λ_{fc}. Assume in particular that $\lambda_{fc} * G_{fc}(0.16) = \lambda_{sl} \Rightarrow 0.25 * \lambda_{fc} = \lambda_{sl}$; $\lambda_{fc} * G_{fc}(0.12) = \lambda_{lb} \Rightarrow 0.10 * \lambda_{fc} = \lambda_{lb}$; and $\lambda_{fc} * G_{fc}(0.12) = \lambda_{co} \Rightarrow 0.10 * \lambda_{fc} = \lambda_{co}$. We substitute the values of λ_{cr}, λ_{sl}, λ_{lb},andλ_{co} into $\sum \lambda_i = 1$. After solving the equations obtained simultaneously, we get $\lambda_{fc} = 0.51$; $\lambda_{cr} = 0.26$; $\lambda_{sl} = 0.127$; $\lambda_{lb} = 0.05$; and $\lambda_{co} = 0.05$.

Usually, the ratio between the values from the preference vector from *Step 4* should give a rough estimate as to the ratio between the different λ_i's, provided all attributes are measured on the same scale. If not, they could be scaled and compared. Having obtained λ_i, and G_i for each attribute, we can compute the goodness function, which in this case will be (from Eq. (8)):

$$G(q_{fc}, q_{cr}, q_{sl}, q_{lb}, q_{co}) = 0.51 \cdot G_{fc}(q_{fc}) + 0.26 \cdot G_{cr}(q_{cr}) + 0.127 \cdot G_{sl}(q_{sl}) + 0.05 \cdot G_{lb}(q_{lb}) + 0.05 \cdot G_{co}(q_{co}) \quad (9)$$

Note that the function is additive, since the attributes are mutually preferentially independent. The values of individual goodness functions corresponding to the value of attributes for each mapping are determined from the graphs of Figs. 15(a)–16(c).

9.7. Step 7: Calculating Overall Function

The *overall goodness* values of the different mappings are (using Eq. (9)):

- $G(V_A) = 0.51 \cdot G_{fc}(0.75) + 0.26 \cdot G_{cr}(0.10) + 0.127 \cdot G_{sl}(0.10) + 0.05 \cdot G_{lb}(0.15) + 0.05 \cdot G_{co}(0.3) = 0.51 * 1 + 0.26 * 0 + 0.127 * 0 + 0.05 * 0 + 0.05 * 0.68 = 0.54$ *(Mapping A)*
- $G(V_B) = 0.51 \cdot G_{fc}(0.15) + 0.26 \cdot G_{cr}(0.80) + 0.127 \cdot G_{sl}(0.18) + 0.05 \cdot G_{lb}(0.75) + 0.05 \cdot G_{co}(0.10) = 0.51 * 0.15 + 0.26 * 1 + 0.127 * 0.50 + 0.05 * 1 + 0.05 * 0 = 0.45$ *(Mapping B)*
- $G(V_C) = 0.51 \cdot G_{fc}(0.10) + 0.26 \cdot G_{cr}(0.15) + 0.127 \cdot G_{sl}(0.30) + 0.05 \cdot G_{lb}(0.25) + 0.05 \cdot G_{co}(0.5) = 0.51 * 0 + 0.26 * 0.24 + 0.127 * 1 + 0.05 * 0.40 + 0.05 * 1 = 0.26$ *(Mapping C)*

Fig. 17 shows the overall goodness profile of the three mappings and illustrates how the different attributes contributed to the overall goodness value. Note that emphasizing other attributes than in this example may result in different goodness profiles.

9.8. Step 8: decision making on the best alternative

We select *Mapping A* as it results in the highest goodness value, when dependability is our main concern. This is consistent given that the heuristic used to generate *Mapping A* is dependability-oriented. Hence, the final mapping profile is: $V_A = (0.75, 0.10, 0.10, 0.15, 0.30)$.

9.9. Step 9: refinement

Step 9 characterizes the overall *refinement* allowing for refinement of the goodness function for improving the trade-off provisioning. In this example, the trade-off study is not really appropriate as we have three alternatives which are far apart attribute-wise. Better illustration of trade-offs would be achieved, if heuristics H1 or H2 were used to generate the mappings. However the inherent intractability of obtaining a precise goodness function may warrant several iterations through the decision procedure for refinement. The aspiration vector is reassessed and *S*4 to *S*9 re-executed. Care should be taken so that the newly generated goodness function closely reflects the changes made. Based on these changes and on the goodness value of the new mapping (if applicable at all), trade-offs can be better ascertained. This step also enables the decision maker to reflect on the solution at hand and

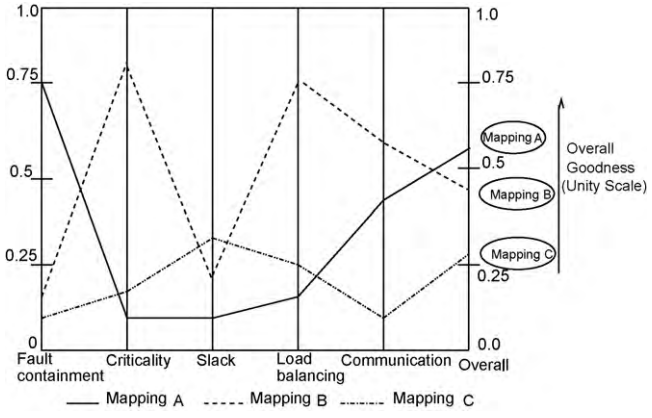

Fig. 17. Performance/goodness profiles for the three mappings.

allows him/her to consider making some trade-offs, by readjusting the preference structure, i.e., it enables an interactive and iterative decision procedure.

10. Applicability and utility of the 9-step decision procedure

Designing integrated dependable systems entails mapping the fault containing processes to HW and choosing the best available mapping. This implies using a framework able to handle the conflicting requirements of processes. Decision theory provides a basis upon which trade-offs can be resolved so that the best available option is derived, via an optimization function, termed here as *goodness* function. Trade-offs are arbitrated through the system requirements that are encoded in the goodness function.

We thus introduced two techniques for assessing the goodness of mappings. The first, *random exploratory technique*, is a search-based technique that systematically probes the effectiveness frontier (Pareto optimal set) of the mapping set. The second technique is a *9-step interactive decision procedure* that systematically guides the designer in formalizing the system requirements. This allows determination of the goodness function that encapsulates these requirements as the goodness function of individual attributes (G_i) and the importance of individual attributes (λ_i). This function has the property of creating a weak preference ordering among the different mappings, such that the best available mapping is the one with the highest goodness value. Value trade-offs are thus easily resolved in this framework.

The real novelty of this approach is the creation of an interpolation surface whereby trade-offs can be accounted for. We are currently investigating different techniques of generating different interpolation surfaces that make trade-offs more explicit, i.e., instead of looking at midpoints for a single attribute, one can look at pairs of attributes in such a way that trade-offs are more explicit. We also assumed evaluator functions to obtain attribute vectors. One aspect currently under investigation is determining the attribute evaluators that work best for dependable embedded systems.

However, a good (best) mapping may still not satisfy the requirements of the system, for inaccurate parameters, i.e., λ_i and G_i are not fully accurate, since generation of a precise goodness function may be intractable. To address this issue, the decision procedure is iterative, such that current mapping information can be reused to perform preference refinement. Hence, step 9 allows refinement to be performed, forcing the designer to re-assess system requirements. The remaining steps force the designer to assess decisions within the given framework.

Overall, this decision procedure has the property of making the trade-offs explicit. Depending on the nature of the integration (dependability-driven, performance-driven, etc.), different mappings may be chosen as the best one (i.e., the goodness profiles of the mappings will be different). In this paper, given the dependability focus, a dependability-driven mapping (*Mapping A*) was selected as the best one. However, if criticality was the more important attribute, *Mapping A* may not have been the best one. Also, had the integration process required that both dependability and criticality be of primary importance, the decision procedure would arbitrate using requirements information pertaining to the other attributes to obtain the best available mapping. A trade-off analysis among the mappings generated by heuristics **H1** and **H2** would help determine the better heuristic for dependability-driven SW integration.

On the other hand, the main limitations of this 9-step decision procedure are (i) to get accurate results, the decision procedure needs to be iterated several times until a proper preferential structure is obtained that will result in a more accurate goodness function, and (ii) to obtain the graph of the different single-attribute functions (as well as the *aspiration vector*), in-depth knowledge of system requirements is needed. Also, there may be a need for several designers to provide the aspiration vector such that inconsistencies be filtered out at an early stage. This will help in determining more accurate single-attribute functions.

11. Summary

Our approach for developing integrated dependable SW made the following contribution: (a) formulation of a hierarchical structure for partitioning of SW modules, (b) composition strategies for creating integrated SW modules, (c) quantification of interaction between SW modules, and (d) development of techniques of mapping SW modules onto HW, (e) proposing the random exploratory technique, and (f) introduction of a decision procedure for determining the goodness of mappings. Specifically,

- We started with a standard object-oriented design which may or may not have been designed with dependability as main focus. We decomposed the object-oriented design into smaller modules, which are then transformed, if necessary, into FCMs. These FCMs have the property that they contain errors with high probability within their boundary at any abstraction level.
- Using the vertical (global) composition strategies, bigger modules were obtained while the horizontal (local) integration (influence) deals with indication of error propagation among FCMs to detect any vulnerabilities.
- Once process FCMs were obtained (whose fault containment capabilities are well-defined at this point), they were mapped onto HW nodes. To achieve this, heuristics based systematic mapping process has been provided.
- Given that different heuristics will give rise to different mappings, to obtain the mapping that best meets the requirements of the system, we proposed two decision procedure techniques. The first, *Random Exploratory Technique*, is analogous to a search procedure. The second was a formal decision procedure, derivable from system requirements.

Overall, the framework presented here is not dependent on the number of levels of hierarchy introduced by the decomposition process. Rather, one crucial aspect is evaluation of influences between FCMs at each level. Also, another crucial point is the evaluation of the different mappings obtained from different clustering algorithms. Thus, the impact of different clusters on the integration process can be visualized using the different profile obtainable from our framework. The best mapping identified through the profiling technique is the one that satisfies both hardware constraints, and also dependability and other associated constraints, such as load balancing, etc. In the future, we plan to address relative trade-offs between approaches with more detailed models to include domain/application-specific trade-offs. It is also of interest to develop a trade-off analysis across HW and SW requirements, especially when design restrictions are provided on the choice of available HW platform.

Acknowledgments

The authors would like to thank S. Winter for his assiduous work on earlier versions of this paper. Also, S. Ghosh's contributions and discussions on Sections 1–7 are highly appreciated. Similarly, A. Balogh significantly contributed to the improvement of Section 2.

References

Ahuja, R.K., Magnanti, T.L., Orlin, J.B., 1993. Network Flows: Theory, Algorithms and Applications. Prentice Hall.

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1 (1), 11–33.
Baldin, D., Kerstan, T., 2009. Proteus, a hybrid virtualization platform for embedded systems. Journal of Analysis, Architectures and Modelling of Embedded Systems, 185–194.
Chillarege, R., Biyani, S., Rosenthal, J., 1995. Measurement of failure rate in widely distributed software. In: International Symposium on Fault-Tolerant Computing, p. 424.
Cristian, F., Aghili, H., Strong, R., Dolev, D., pp. 200–206 1985. Atomic broadcast: from simple message diffusion to byzantine agreement. In: Information and Computation.
Czyzak, P., Jaszkiewicz, A., 1998. Pareto simulated annealing–a metaheuristic technique for multiple-objective combinatorial optimization. Journal of Multi-Criteria Decision Analysis 6 (7), 34–47.
David, R., 1998. Random Testing of Digital Circuits: Theory and Applications. Marcel Dekker.
Ekelin, C., Jonsson, J., 2001. Evaluation of search heuristics for embedded system scheduling problems. In: International Conference on Principles and Practice of Constraint Programming. Springer-Verlag, London, UK, pp. 640–654.
Flanagan, D., 1999. Java in a Nutshell: A Desktop Quick Reference. O'Reilly & Associates, Inc., Sebastopol, CA, USA.
Garey, M.R., Johnson, D.S., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co, New York, NY, USA.
Ghosh, S., Rajkumar, R.R., Hansen, J., Lehoczky, J., 2003. Scalable resource allocation for multi-processor qos optimization. In: International Conference on Distributed Computing Systems. IEEE Computer Society, Los Alamitos, CA, USA, pp. 174–185.
Hiller, M., 2000. Executable assertions for detecting data errors in embedded control systems. In: DSN, pp. 24–33.
Hiller, M., Jhumka, A., Suri, N., May 2004. Epic: profiling the propagation and effect of data errors in software. IEEE Transactions on Computers 53 (5), 512–530.
Horvitz, E.J., Breese, J.S., Henrion, M., 1988. Decision theory in expert systems and artificial intelligence. International Journal of Approximate Reasoning 2 (3), 247–302.
Islam, S., Lindström, R., Suri, N., 2006. Dependability driven integration of mixed criticality sw components. In: ISORC, pp. 485–495.
Islam, S., Suri, N., 2007. A multi variable optimization approach for the design of integrated dependable real-time embedded systems. In: International Conference on Embedded and Ubiquitous Computing, vol. 4808, pp. 517–530.
Islam, S., Suri, N., Balogh, A., Csertán, G., Pataricza, A., 2009. An optimization based design for integrated dependable real-time embedded systems. Journal of Design Automation for Embedded Systems 13 (4), 245.
Ismaeel, A.A., Breuer, M.A., 1991. The probability of error detection in sequential circuits using random test vectors. Journal of Electronic and Testing 1 (4), 245–256.
Izosimov, V., Pop, P., Eles, P., Peng, Z., 2005. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In: DATE. IEEE Computer Society, Washington, DC, USA, pp. 864–869.
Jhumka, A., Hiller, M., Claesson, V., Suri, N., 2002a. On systematic design of globally consistent executable assertions in embedded software. In: LCTES-SCOPES, pp. 75–84.
Jhumka, A., Hiller, M., Suri, N., 2001. Assessing inter-modular error propagation in distributed software. In: SRDS, pp. 152–161.
Jhumka, A., Hiller, M., Suri, N., 2002b. An approach to specify and test component-based dependable software. In: HASE, pp. 211–220.
Jhumka, A., Hiller, M., Suri, N., 2002c. Component-based synthesis of dependable embedded software. In: FTRTFT, pp. 111–128.
Jhumka, A., Klaus, S., Huss, S.A., 2005. A dependability-driven system-level design approach for embedded systems. In: DATE, vol. 1. IEEE Computer Society, Los Alamitos, CA, USA, pp. 372–377.
Kandasamy, N., Hayes, J.P., Murray, B.T., 1999. Tolerating transient faults in statically scheduled safety-critical embedded systems. In: SRDS, vol. 0. IEEE Computer Society, Los Alamitos, CA, USA, p. 212.
Kaufman, L.M., Johnson, B.W., Dugan, J.B., 2002. Coverage estimation using statistics of the extremes for when testing reveals no failures. IEEE Transactions on Computers 51, 3–12.
Keeney, R., Raiffa, H., 1993. Decisions with Multiple Objectives: Preferences and Value Tradeoffs. Cambridge University Press.
Kodase, S., Wang, S., Gu, Z., Shin, K.G., 2003. Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. In: Real-Time and Embedded Technology and Applications Symposium, IEEE 0, p. 181.
Kuchcinski, K., 2003. Constraints-driven scheduling and resource assignment. ACM Transactions on Design Automation of Electronic Systems 8 (3), 355–383.
Lee, Y.-H., Kim, D., Younis, M., Zhou, J., McElroy, J., 2000. Resource scheduling in dependable integrated modular avionics. In: DSN. IEEE Computer Society, Washington, DC, USA, pp. 14–23.
Lipari, G., Carpenter, J., Baruah, S., 2000. A Framework for Achieving Inter-Application Isolation in Multiprogrammed, Hard Real-Time Environments.
Locke, C.D., Vogel, D., Mesler, T.J., 1991. Building a predictable avionics platform in ada: a case study. In: Proceedings of Real-Time Systems Symposium, pp. 181–189.
Luce, R., Krantz, D., Suppes, P., Tversky, A., 1990. Foundations of measurement 3: Representation, axiomatisation and invariance.
Mohanty, S., Prasanna, V.K., Neema, S., Davis, J., 2002. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. SIGPLAN Notices 37 (7), 18–27.
Mustafiz, S., Kienzle, J., 2004. A survey of software development approaches addressing dependability. In: FIDJI, pp. 78–90.
Neema, E., Sztipanovits, J., Karsai, G., 2003. Constraint-based design-space exploration and model synthesis. In: Proceedings of EMSOFT, vol. 2855 of LNCS. Springer, pp. 290–305.
Oh, Y., Son, S.H., 1994. Enhancing fault-tolerance in rate-monotonic scheduling. Real-Time Systems 7 (3), 315–329.
Prasad, D., McDermid, J., 1999. Dependability evaluation using a multi-criteria decision analysis procedure. Dependable Computing for Critical Applications 7, 339–358.
Rajkumar, R., Lee, C., Lehoczky, J.P., Siewiorek, D.P., 1998. Practical solutions for qos-based resource allocation problems. In: IEEE Real-Time Systems Symposium, pp. 296–306.
Randell, B., 1975. System structure for software fault tolerance. In: Proceedings of the international conference on Reliable Software. ACM, New York, NY, USA, pp. 437–449.
Rushby, J., 1999. Partitioning for safety and security: requirements, mechanisms, and assurance (CR-1999-209347).
Saib, S.H., 1978. Executable assertions - an aid to reliable software. In: Proceedings of the 11th Asilomar Conference Circuits Systems and Computer, pp. 277–281.
Stankovic, J.A., Spuri, M., Natale, M.D., Buttazzo, G., 1994. Implications of classical scheduling results for real-time systems. IEEE Computer 28, 16–25.
Suri, N., Ghosh, S., Marlowe, T.J., 1998. A framework for dependability driven software integration. In: ICDCS, pp. 406–415.
Totel, E., Blanquart, J.-P., Deswarte, Y., Powell, D., 1998. Supporting multiple levels of criticality. In: International Symposium on Fault-Tolerant Computing, vol. 0. IEEE Computer Society, Los Alamitos, CA, USA, p. 70.
Wang, S., Merrick, J.R., Shin, K.G., 2004. Component allocation with multiple resource constraints for large embedded real-time software design. In: Real-Time and Embedded Technology and Applications Symposium, vol. 0. IEEE Computer Society, Los Alamitos, CA, USA, p. 219.
Wind River, 2010. Wind River Hypervisor. http://www.windriver.com/products/hypervisor/.
Yin, X., Kiskis, D.L., Mihalik, D., Shin, K.G., 2006. Integration of an analysis tool for large-scale embedded real-time software into a vehicle control platform development tool chain. In: ESA, pp. 53–59.
Younis, M.F., Aboutabl, M., Kim, D., 2004. Software environment for integrating critical real-time control systems. Journal of systems Architecture 50 (11), 649–674.
Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C.M., Grunert da Fonseca, V., 2003. Performance assessment of multiobjective optimizers: an analysis and review. IEEE Transactions on Evolutionary Computation 7 (2), 117–132.

Neeraj Suri is a Chair Professor at TU Darmstadt, Germany at the DEEDS Group in the Dept. of Computer Science. His professional details are available at www.deeds.informatik.tu-darmstadt.de/suri.

Arshad Jhumka obtained his PhD from TU Darmstadt, DEEDS Group. Currently he is a Reader at the Univ. of Warwick, UK. Details are available at www.dcs.warwick.ac.uk/people/academic/Arshad.Jhumka/.

Martin Hiller obtained his PhD from the DEEDS Group. He is currently a Product Area Manager - Embedded Architecture at Volvo Technology Corporation, Sweden.

András Pataricza is a Professor at Budapest University, Hungary. His professional details are available at www.mit.bme.hu/ pataric/.

Shariful Islam obtained his PhD from TU Darmstadt, DEEDS Group. He is currently at Innoventis GmbH, Germany.

Constantin Sârbu obtained his PhD from TU Darmstadt, DEEDS Group. Currently he is a post-doctoral fellow at TU Darmstadt http://www.deeds.informatik.tu-darmstadt.de/dinu/index.html.