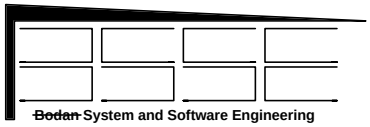
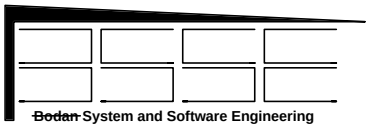
GIFTBox: A Guide to Master Distributed, Heterogeneous Systems

Rainer Gerlich
~~Bodan~~ Software and System Engineering
Auf dem Ruhbuehl 181
D-88090 Immenstaad
Phone: +49/7545/911.258, +49/7545/529
Mobile: +49/171/80.20.659
Fax +49/7545/911.240
e-mail: gerlich@t-online.de





GIFTBox: A Guide To Master Distributed, Heterogeneous Systems

Rainer Gerlich
~~Bodan~~ System and Software Engineering
BSSE
Auf dem Ruhbuehl 181
D-88090 Immenstaad, Germany
Phone: +49/7545/911.258, Mobile: +49/171/80.20.659, Fax +49/7545/911.240
e-mail: gerlich@t-online.de

Abstract: GIFTBox (Generic Interfaces and Fault-Tolerant Boxes) is a scheme for description of heterogeneous and/or distributed systems in terms of generic interfaces and fault-tolerant modules (boxes). This scheme allows to construct a larger system by simple and clear rules out of generic elements ("atoms" and "binding" rules) and provides the capabilities needed to master the problems of maintenance and evolution together with distributed and heterogeneous systems. It is based on a clear and simple classification of communication and system activities. The structuring and communication mechanisms imply fault encapsulation and known error propagation, respectively. The proposed decomposition scheme is an enhancement of the class concept well-known from object-oriented programming (OOP), e.g. using HOOD, Ada or C++. Whilst OOP is just recommending class usage, GIFTBox advises a system engineer which classes he should build or reuse, which functionality shall be included in a box (class), and how composite boxes (higher classes) shall be constructed from basic classes. Vice versa, this construction principle allows to cross-check a system decomposition for forgotten functionality. The generic, but well defined interfaces and the clear control flow allow to integrate existing components using the same or another hardware or software platform. Moreover, the GIFTBox scheme allows to organise and manage a project in accordance with the technical approach. GIFTBox is complementary to the EaSyVaDe approach which aims to reduce development risks by early system validation.

Keywords: System architecture, distributed systems, heterogeneous platforms, system integration, fault-tolerance, maintenance, system evolution, project management

1. INTRODUCTION

Development of larger systems is a challenge because one has to master the contractual, management and technical aspects. This becomes even a higher challenge for distributed, heterogeneous systems which need to be fault-tolerant and open for future evolution.

Contractual aspects deal with the distribution of work over several contractors and the synthesis of a lot of deliverables to the desired system right in time and within the given budget. Management aspects address the strategy of system development, which development approach shall be applied, which commercial products shall be used, how sufficient efficiency can be achieved, how the technical problems shall be tackled, how the system is made sufficiently maintainable and open for evolution, how the required quality can be achieved. The technical aspects concentrate on a system's operations, on their implementation by hardware and software within given environmental constraints.

At the first glance all the three types of aspects seem to be independent and it is neither obvious that they can be co-ordinated nor that there is a need to harmonise them. Work packages, a contractual matter, are defined in terms of development resources, like man power, facilities and time. Why shall they be correlated with development strategy or implementation of system functions? Such aspects can be considered in each work package internally.

However, such a correlation becomes evident when something goes wrong. If we have to make a technical change what is the contractual implication? Is only one contractor effected or a number of contractors? Then we may recognise that by harmonisation of contractual and technical aspects we would have produced less effort.

Similarly it is for contractual and management aspects. E.g. reuse is not only a matter inside a contract. Several contracts may include similar functionality. Why not to reuse components across such contracts? However, this has to be organised.

Also, between management and technical aspects a dependency exists. E.g. maintainability and testability are management issues, but they are depending on the technical implementation.

So each type of aspects is in a potential conflict with the other ones. In such a case one GIFTBox rule (which is explained later) applies already: whenever there are competing elements (slaves) they need a co-ordinator (master). In this case GIFTBox is the co-ordinator itself. It allows to harmonise the potentially conflicting goals of each aspect.

GIFTBox puts main emphasis on the technical aspects under consideration of the management and contractual aspects. It concentrates on system operations because this is what is expected later as service from a system. When the system is in operation we do not see any more contractual boundaries or management issues. They have to disappear completely. If not, system operations would be compromised.

GIFTBox solves the potential conflict between contractual and technical aspects by encapsulation of an operational sub hierarchy of the system into a contract[1]. The management aspects are considered by introduction of clear generic rules which ensure good maintainability, testability and openness for future evolution. Higher degree of reuse is introduced by standardisation of interfaces and guidelines for identification of reusable components. To summarise, the potential conflicts are solved by organising the technical approach such that the contractual and management matters are well covered.

GIFTBox assumes the worst case of system development: heterogeneous, distributed systems, need for integration of existing (sub-)systems, maintainable for a number of years, open for future evolution. In the following chapters GIFTBox is briefly described, and its relevance for fault-tolerant, heterogeneous and distributed systems is discussed.

2. GIFTBOX GUIDELINES

GIFTBox has already been considered from different points of view by two other papers [1,2]. In [1] it is used for system refinement when applying the EaSyVaDe methodology (EaSyVaDe = Early System Validation of Design). This methodology asks for incremental system validation in order to reduce the risks at the earlier life cycle phases. In [2] the benefit of formalisation is discussed and GIFTBox is used to ease integration and to increase reuse. The current paper addresses relevance of GIFTBox to distribution, heterogeneous platforms and fault-tolerance.

The GIFTBox guidelines and associated figures shall briefly be repeated and explained here:

1. *generic interfaces*
   they define the types of communication between generic fault-tolerant boxes and the semantics of the types (Fig. 2-1)
2. *master-slave concept*
   a co-ordinator ("master") is needed for a set of boxes (components, modules, "slaves") appearing on the same level to avoid conflicts in responsibility between the slaves (Fig. 2-2)

[1] Several sub hierarchies could be put under the same contract, if needed. But either such sub hierarchies have the same root on the next higher level, then we get again only one sub hierarchy, or they are independent and do not interfere, so one contract covers several subcontracts.

3. *top-down decomposition*[2]
   a. decomposition into a generic and an application specific box (Fig. 2-3)
   b. identification of dynamic components (modes) (Fig. 2-4)
   c. identification of static components (operations, functions, services, subsystems) (Fig. 2-5)
   d. iteration between static and dynamic levels (Fig. 2-6)
   e. decomposition of the generic manager into principal management components (Fig.2-7)

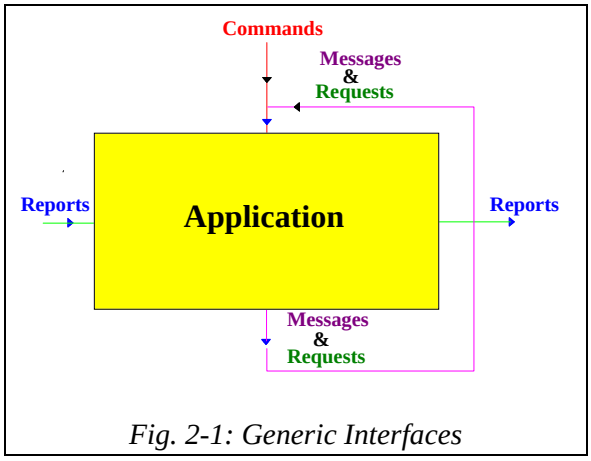


Fig. 2-1: Generic Interfaces

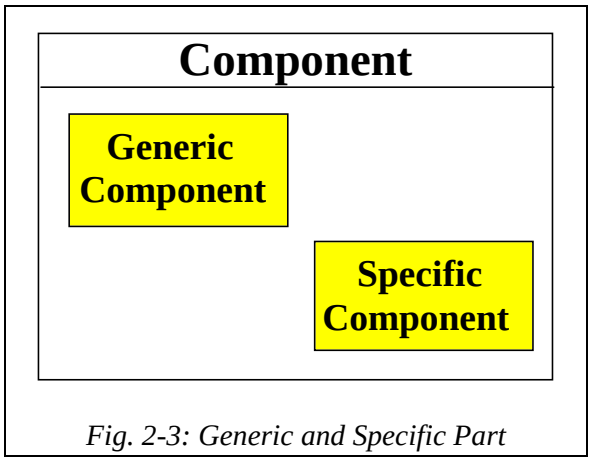


Fig. 2-3: Generic and Specific Part

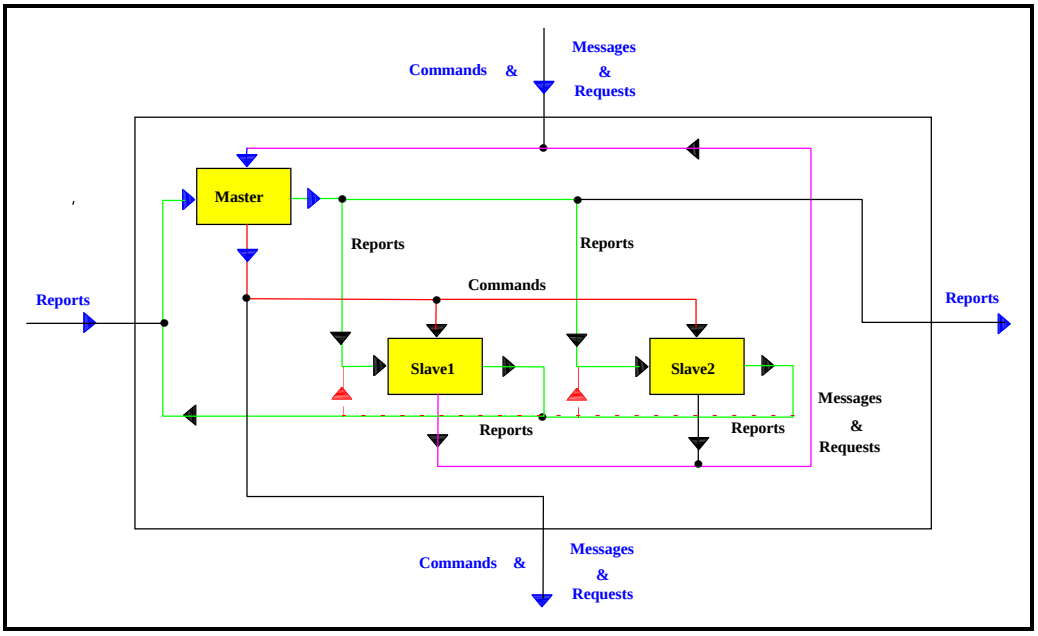


Fig. 2-2: Control Hierarchy and Master-Slave Concept

[2] In reverse order ("bottom-up" synthesis) the related rules can also be applied, of course.

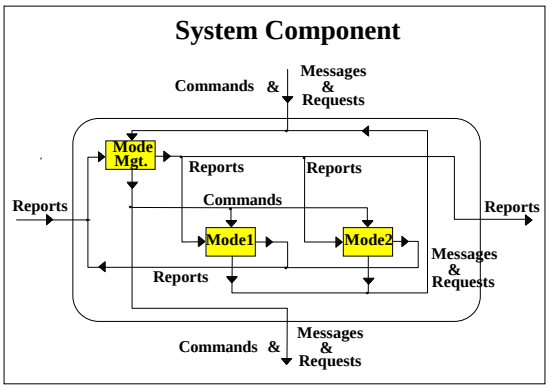


Fig. 2-4: System Component and Its Dynamic Internals

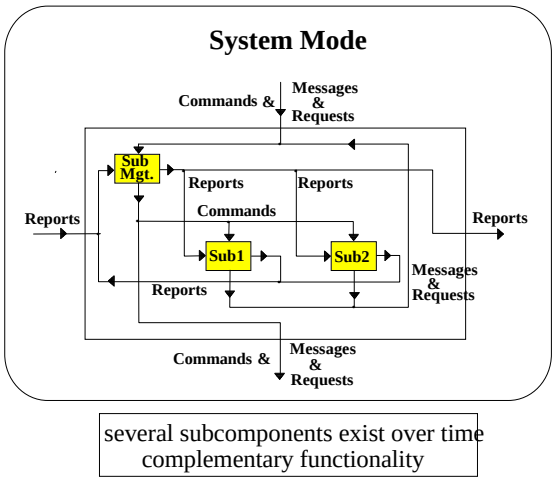


Fig. 2-5: System Mode and Its Static Internals

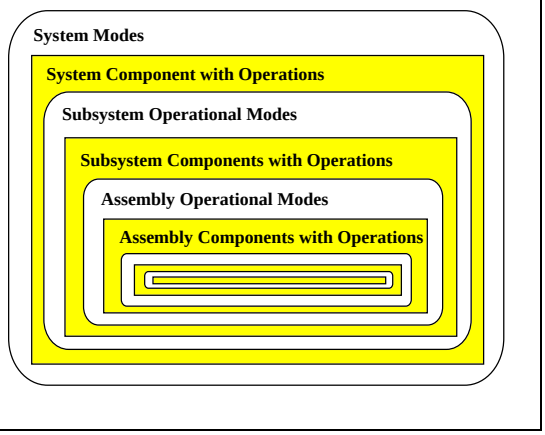


Fig. 2-6: Iterations between Components: Operations and Modes

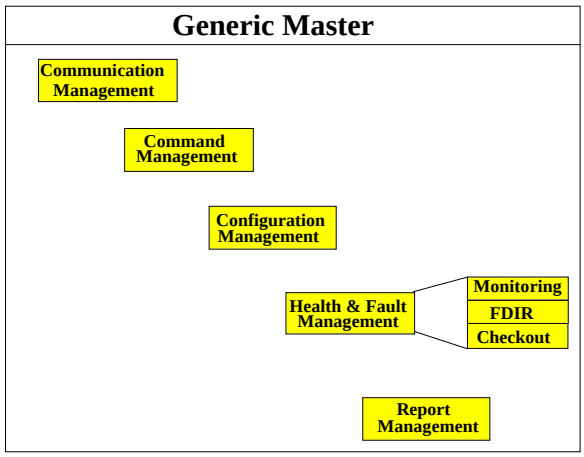


Fig. 2-7: Principal Management Components

The interface types (Fig. 2-1) are divided into *control flow* (from top to bottom) and *data flow* (from left to right). Data are not further subdivided, they are just treated as *reports* which are results of processing and are used as input to another component. Control flow is subdivided into *commands*, *messages* and *requests*. The structure of a component's interface is fixed by the shown interface lines.

By using the fixed interface structure and the well-defined semantics we are able to build a component's internals in a clean manner, understanding what is coming in and what is leaving, and what is the impact on a component.

By commands a master *advises* a slave to follow directly the given instructions. There is no way to reject except in contingency case. A command may change the mode[3] of the slave. By a request a

[3] A *mode* is related to a certain operational goal a component aims to achieve during a certain period of time. This goal may be changed according to the mission phases. The services needed to achieve such a goal may be different during different modes. A mode switch may be needed e.g. in case of a fault if only another mode allows to keep the system alive.

slave asks (politely) his master for some services, e.g. for information (reports). The master may not directly respond to the request. A request (normally) does not change the mode of the master. By a message a slave informs his master about his state, e.g. by telling 'command execution started' which is a positive message. However, negative messages are also possible like 'performance lost', 'command execution failed'.

Basically, a slave must not change the state of his master. But in case of such negative messages *fault propagation* occurs: a (fault-tolerant) box failed to provide the expected service. In this case a master may have to change his mode, because he cannot continue his service. He may try to activate a redundant component of a slave in order to recover from this fault, but he may not be able to recover. If no complete recovery from the fault is possible the fault propagates again to the next higher level, possibly with less degree of severity.

The *master-slave* concept (Fig. 2-2) is introduced in order to get a clean approach for responsibilities and commanding. Also, the master represents the interface to the outside world. If the contents of an interface is changed (the structure is fixed according to rule 1) then the slaves are not (necessarily) effected. Vice versa if internal interfaces are changed the outside world is not effected because the master absorbs such changes.

Each slave only communicates with his master concerning commands, messages and requests. However, another slave may listen to the reports sent by any slave. This allows to implement slaves which act as communication medium (channels). A master may provide to his slaves a communication mechanism to the outside world in a transparent manner (see also Figs. 4-1 and 4-3 below).

It was found that certain management tasks are needed at several places in the whole hierarchy (Fig. 2-3). In order to increase reuse a master is subdivided into a *generic* and an *application specific* part.

When we look at a system component we first see its different modes (Fig. 2-4), e.g. power or de powered, standby or fully operational. For each mode a certain set of operations is provided by sub components (Fig. 2-5). In order to manage a *mode transition* in a clean manner we need a *mode manager* which acts according to the master-slave rule (GIFTBox rule 2). He checks if the conditions are fulfilled to enter or to leave a certain mode.

The system components related to each mode are also managed by the master-slave concept as shown by Fig. 2-5. In this manner a system is built like an onion consisting of *mode shells* and *operational shells* (Fig. 2-6): a mode shell follows an operational shell and vice versa[4].

The principal components of a *Generic Component* are shown in Fig. 2-7. What is common to each principal component is communication management, command processing, configuration management (e.g. to recover from a fault by reconfiguration), health and fault management and report management. Monitoring of limits or of other properties is a generic functionality which can be specialised by data. FDIR (Fault Detection, Identification and Recovery) may also be expressed by generic functions. By checkout non-operational components are tested whether they are faulty or not before they shall become operational. Report management addresses the collection of data and their formatting. Communication management may provide directly the means for command and report transfer to each slave. Then a slave can directly transmit data to the outside world still not knowing what the communication mechanism really is. The same communication resources may be shared by several masters in the hierarchy.

Our experience is that GIFTBox rules 1 -3 apply in every case. E.g. if we did not foresee a master because we were convinced there is nothing to do for a master, later on it turned out that there *is*

[4] Within each such shell of an "onion" (box) a number of onions (boxes) can occur again as shown by Fig. 2-2.

something to do. In consequence, we decided to do it just vice versa: we assume that a master is needed and we look for the tasks he has to do, and we succeeded with this approach.

3. RELATIONSHIP TO EASYVADE

GIFTBox yields a hierarchical decomposition. By EaSyVaDe[5] a system is also hierarchically decomposed. During system refinement the impact on the hierarchy by extensions or changes in a certain branch must be minimised, e.g. in case of a transition between specification and design or in case of iterations. GIFTBox is the scheme which allows to master this challenge of keeping the hierarchy as stable as possible. Basically, we had the GIFTBox scheme in mind when we defined the EaSyVaDe life cycle approach.

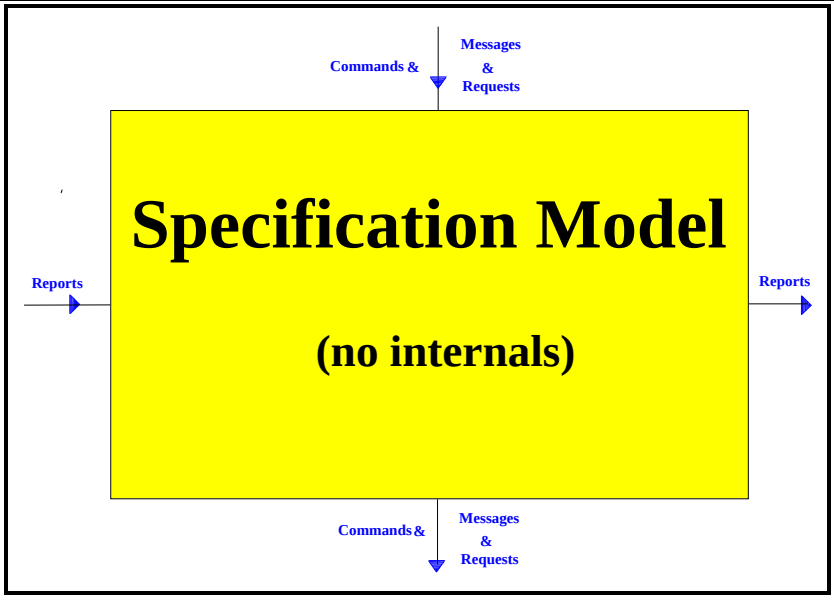


Fig. 3-1: Specification Box: A High Level View

EaSyVaDe uses executable models during specification and design from which target code is derived. Each such model is represented by a GIFTBox box.

When making the transition from specification to design the interface of the specification model (specification box) is kept for the design model (design box). This transition means to give the specification model of Fig. 3-1 an internal structure as shown by Fig. 2-2: the simple specification box is expanded into the more detailed design box according to the refinement of functionality, behaviour and the introduced architecture. The master inside a design box limits the impact by external changes onto the internal components. Similarly, internal changes are limited to the outside world.

4. TOWARDS FAULT-TOLERANCE

The degree of automation will grow in future and therefore fault tolerance is becoming more and more important. Final decisions made currently by a human operator may be performed automatically by a system in future.

In case of distributed and heterogeneous systems we have to care about data consistency and availability of data right in time, and to prevent commanding and access conflicts due to insufficient communication and co-ordination means.

---

[5] EaSyVaDe is an outcome of the ESTEC project OMBSIM (On-Board Management System Behavioural Simulation) [3]. It is now used for the ESTEC study DDV (Data Management System Design Validation) [4]. An incremental life cycle approach based on EaSyVaDe is described in [5]. Results of application are given in [6a, 6b].

For implementation of fault-tolerance one needs (a) to define exactly the component which shall be fault-tolerant and (b) to care about what shall happen if an error[6] propagates (for whatever reasons) outside such a fault-tolerant component.

Again, GIFTBox harmonises the issue of fault-tolerance with the issue of getting clear operational interfaces: a fault-tolerant component (FTC) is identical with a sub hierarchy, and a generic GIFTBox interface defines exactly not only the data and command flow, but also the flow of exceptions (the *messages* of Fig. 2-1). This allows to encapsulate not only the functionality and behaviour but also error recovery. An error may propagate bottom-up until it is handled[7]. If it cannot be handled then the component needing the lost service has to switch to a degraded mode. The mode manager of Fig. 2-4 allows to perform such a mode transition in a controlled manner.

One achieves fault-tolerance (a) by asking for each service visible at an interface what happens if it is lost, and (b) by providing a recovery procedure in the component using this service. Consequently, one has to care about loss of services or not getting it right in time. Whenever this is forgotten or not completely done one will loose system performance if an error occurs in the related part.

The benefit of such recovery procedures has to be compared with costs of their implementation, of course. So the finally implemented degree of fault tolerance is a compromise between minimum system performance which can be tolerated and available budget. Such considerations may lead to a decision that not each single service has to be fault-tolerant but the whole module providing the service.

Also, there might be cases for which it is only possible to handle a failure sufficiently at a higher level. Then it does not make sense to provide fault-tolerance at the level of fault occurrence. In such cases after any failure a redundant module has to be available rather than a redundant service. Hence a fault may propagate bottom-up until it will be handled at a higher level.

So fault propagation may be a reasonable part of a fault-tolerant concept and not in conflict with the idea of fault-tolerance provided that fault recovery is ensured before the system looses its minimum set of required services.

GIFTBox guides an engineer how to implement fault tolerance by making the interfaces visible: for each service provided by a lower level component he may implement a recovery action, at least he has to make up his mind whether to provide it or not. To recover from an error one must initiate a *controlled* switch to the backup (see Fig. 2-4). GIFTBox advises an engineer to foresee such detection and recovery mechanisms in each (major) component by the rules and guidelines given above in chapter 2.

5. DISTRIBUTED AND HETEROGENEOUS SYSTEMS

Even if we do not think at the beginning to build a distributed, heterogeneous system, we cannot be sure to end up with such a system, when the system will be maintained and will evolve. Also, when we go to build a system we may not know if during development it has to become a distributed system for performance reasons or if we need heterogeneous platforms to cover the desired functionality in an optimum manner.

Consequently, we have to take a development approach which allows to introduce distribution and heterogeneous platforms without major overhead if we need to do it. Vice versa, we should not be charged for something which we do not need. A potential capability should not cause any overhead if we do not need it.

---

[6] The definition for faults, errors and failures is: (a) faults are the source of anomalies, (b) an error is the manifestation of a fault, (c) a failure is the weighted impact of an error on a system.

[7] HOOD and Ada use also such error propagation mechanisms. However, they do not provide rules how to foresee (or not to forget) handling of errors in a systematic manner.

To be flexible enough we need a transparent mechanism for support of distribution and heterogeneous platforms. The driving idea is that system services are a matter of user needs, hence they are invariant (at least after the user requirements have been frozen). Distribution or the type of platforms should not effect the functional decomposition, they are just a means to optimise system performance, to cover environmental constraints or future evolution.

To achieve the de coupling of system capabilities from their implementation we need clean interfaces so that other parts of a system are not effected when we introduce distribution, another platform or when we decide to migrate a system service to another system node in the network. The interfaces and the master-slave concept described above allow to achieve this degree of de coupling

As already mentioned above a master keeps the interfaces between its environment and its internals. When an internal is migrated they are replaced by another component (dummy box) which manages the remote access to the migrated component (Slaves 1 and 2 in Fig. 5-1). Replacement of an internal is equivalent to a change and therefore again the master ensures the stability of his environment.

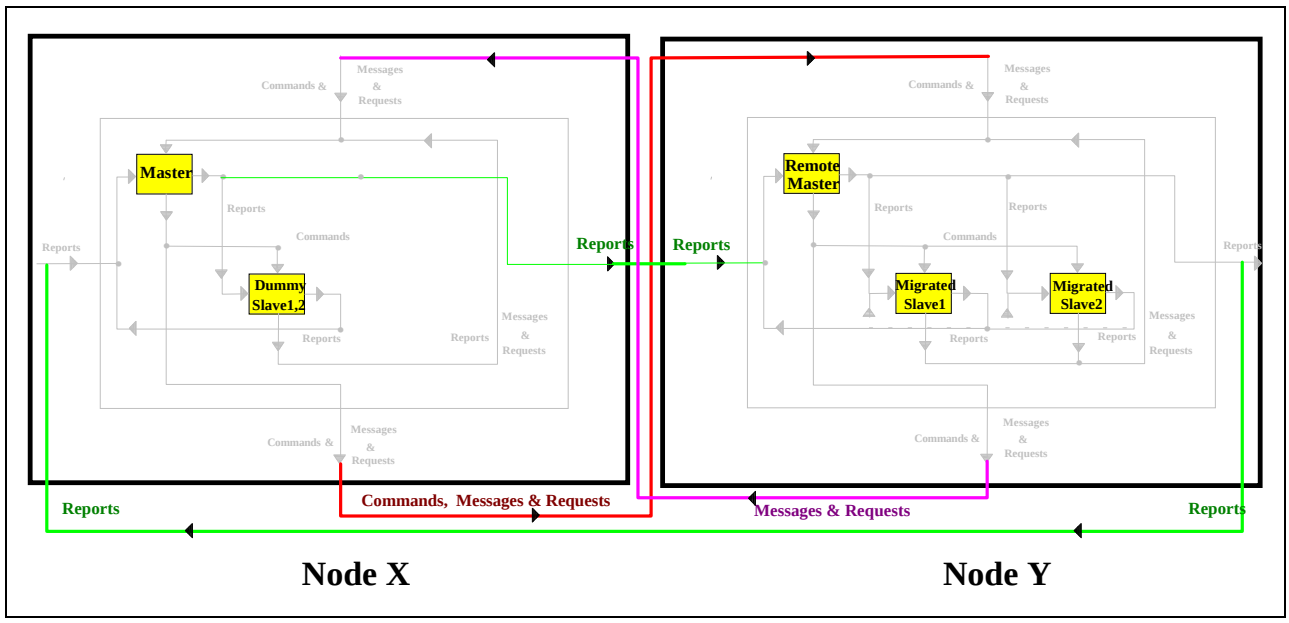


Fig. 5-1: Transparent Migration of System Components

The migrated internal components again get a master at the remote site and are included in a box as before. This master communicates with the dummy box of Node X (via the associated master of the dummy box) so that the communication between a system's components is performed as before. The additional communication link is not visible to components of Node X. It is even not visible for the migrated components themselves. This transparent mechanism for migration and distribution has already been demonstrated [7,8] and used [9]. The new "Virtual Node" concept of HOOD4 [10, 11] is based on this approach.

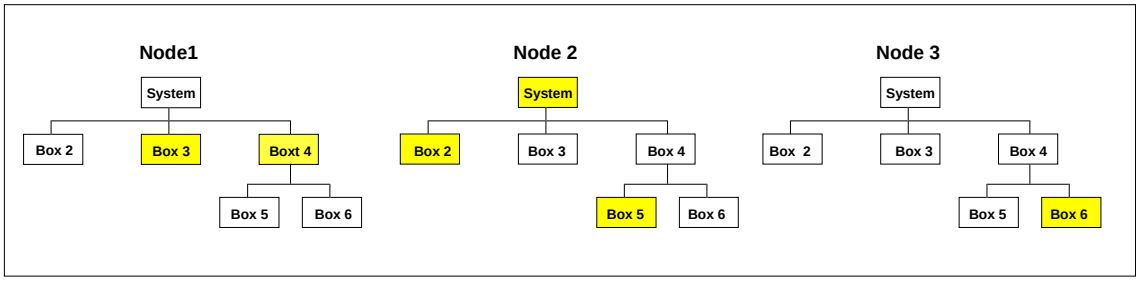


Fig. 5-2: Allocation of System Components to Nodes

It is also possible to migrate only a subset of the slaves or to run sub-components on other nodes than the root components. Fig. 5-2 shows distribution of a system hierarchy over three nodes. In principle, arbitrary allocation of components to nodes is possible. The mapping is just a matter of system performance and environmental constraints.

Introduction of heterogeneous platforms is mastered in the same manner. The transition between platforms is managed by the master interface (Fig. 5-3). The communication link is considered as part of the generic master (Fig. 2-7). Therefore neither external components nor internal ones are effected.

This principle also allows heterogeneous (hard real-time) scheduling of tasks on different platforms (if appropriate from a technical point of view). A scheduler is part of the commanding capability of the generic master. Each master gets a certain time window from the higher level master during which his tasks can be scheduled [10,12].

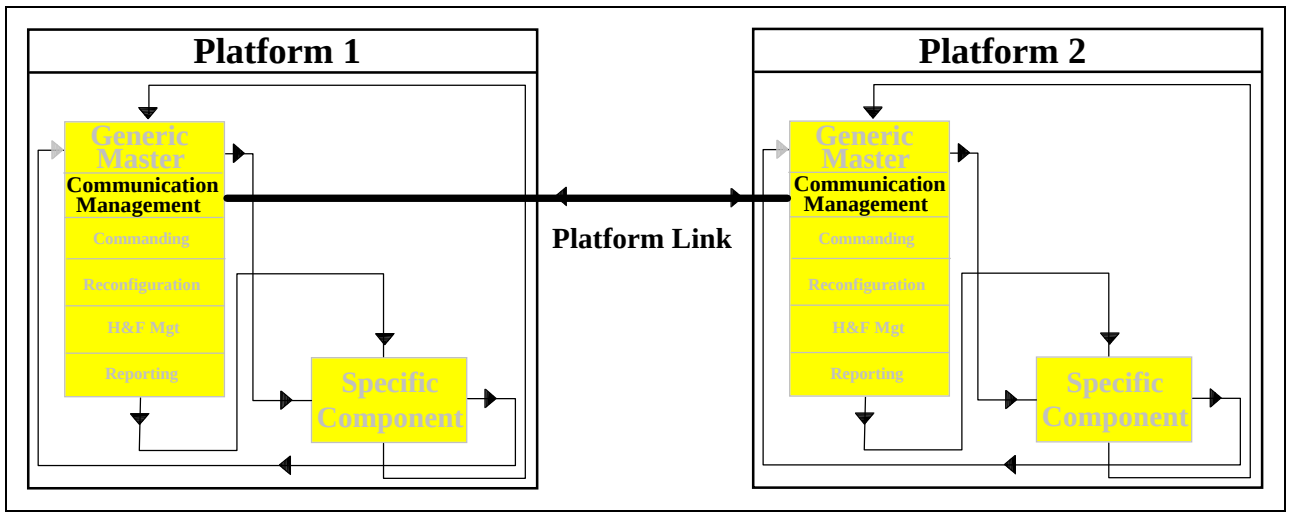


Fig. 5-3: Management of Heterogeneous Platforms

6. CONCLUSIONS

It has been shown that a number of aspects for system development, maintenance and evolution can be mastered by a small set of rules. Emphasis is put on generic interfaces and clear control flow. To avoid commanding conflicts a master-slave relationship is introduced. The master covers principal and generic activities of system components and allows to handle distribution and heterogeneous platforms in a transparent manner. Through the generic interfaces fault-tolerant properties are defined. Handling of errors is supported by a clear commanding concept for mode changes.

The GIFTBox scheme may be applied to a broad range of applications, especially to large and/or safety-critical ones.

6. REFERENCES

[1] R.Gerlich, Ch.Schaffer, Y.Tanurhan, V.Debus: EaSyVaDe / EaSySim: "Early System Validation of Design by Behavioural Simulation", ESTEC 3rd Workshop on "Simulators for European Space Programmes", November 15-17, 1994, Noordwijk, The Netherlands

[2] R.Gerlich: "CIVE (Computer Integrated Validation Environment): A Future Challenge", DASIA96 (Data Management Systems in Aerospace), Eurospace conference, May 20-25, 1996, Rome, Italy

[3] OMBSIM (On-Board Management System Behavioural Simulation, ESTEC contract no. 10430/93/NL/FM(SC), Final Report February 1996, Noordwijk, The Netherlands

[4] DDV (DMS Design Validation), ESTEC contract no. 9558/91/NL/JG(SC), Formal Methods and Tools, Selection & Justification Report, TR/171/PhH/95, 30.06.95

[5] R.Gerlich, C.Joergensen: "An Alternative Life cycle Based on Problem-Oriented Methods and Strategies", International Symposium on On-Board Real-Time Software, ESTEC, Noordwijk, The Netherlands, November 13-15, 1995

[6a] R.Gerlich, Th. Stingl, Ch. Schaffer, F. Teston, G. Martinelli, Y. Tanurhan: "Use of an Extended SDL Environment for Specification and Design of On-Board Operations", Systems Engineering Workshop, November 28-30, 1995, ESTEC, Noordwijk, The Netherlands

[6b] R.Gerlich: "From CASE to CIVE: A Future Challenge!", DASIA'96, Data Systems in Aerospace, May 20-23, 1996, Rome, Italy

[7] R.Gerlich: "On-Line Replacement and Reconfiguration of Ada Real-Time Software", Eurospace Symposium "Ada in Aerospace", December 1990, Barcelona, Spain

[8] R.Gerlich: "Run-Time Linking and On-Line Mode Management with Ada", ESA 1st Conference on Spacecraft Guidance, Navigation and Control, May 1991, Noordwijk, The Netherlands

[9] J.-M. Letteron: "SOFTPAR: A Software Factory for Parallel Applications", ESPRIT Project 8451, Project Report (S29.2), January 11, 1996

[10] HOOD4 Reference Manual, 1996, HOOD User Group, Brussels, Belgium

[11] R.Gerlich, M.Kerep: Parallel and Distributed Systems and HOOD4, Ada in Europe 1995, October 2-6, 1995, Frankfurt, Germany

[12] R.Gerlich: "Proposal for Hard Real-Time Extensions for HOOD4", ESPRIT Project 8451, SOFTPAR, D3.3, July 7, 1995