APPROACHES TO DESIGNING COMPLEX DEPENDABLE SYSTEMS

Andrea Clematis [*,1], **Vittoria Gianuzzi** [**], **Alexander Romanovsky** [***], **Andy M. Tyrrell** [****], **Walter Cazzola** [†]

[*] *IMA - CNR, Via De Marini, 6 - 16149, Genova, Italy*
[**] *DISI, Università, Via Dodecaneso, 35 - 16146, Genova, Italy*
[***] *Dept. of Computing Science - University, Newcastle upon Tyne, NE1 7RU, UK*
[****] *Dept. of Electronics - University, Heslington, York, YO1 5DD, UK*
[†] *DSI Università di Milano, Via Comelico39/41, 20135 Milano, Italy*

Abstract: The problem of designing complex dependable systems is addressed in this paper. Due to some peculiarities of their application and behavior these are often referred to as reactive systems. Two main paradigms for their design have recently been proposed; we name these paradigms living processes and hidden concurrency, depending on their approach to concurrency handling. The analysis of application requirements and constraints is proposed as a methodology for selecting the most suitable implementation paradigm for a given application. Finally, it is shown that in some cases an intermediate paradigm may provide a suitable solution.

Keywords: Fault-tolerant software, Distributed computer control systems, Programming approaches.

1. INTRODUCTION

Computer based systems are used in a wide range of applications including such different fields as automatic banking, factory automation, power plant control, and fly-by-wire systems. There are a lot of differences between all these applications, and each has its own specific requirements; nevertheless, it is possible to find important characteristics common for all these systems.

One common property is that all of them are intended to guarantee the correct functionality of the controlled equipment. The equipment works correctly if it is able to deliver the expected service by properly interacting with the external world. For this reason, the term reactive, or event driven systems, is often used.

Another common characteristic is the complexity of the system. Their design and implementation are never trivial and in some cases are highly complex. One of the main reasons for this complexity is the concurrent nature of reactive systems, i.e. the necessity of keeping track and coordinating two or more concurrent activities.

Yet another common aspect is the general requirement of high dependability, which may involve the use of different types of fault tolerant strategies.

In the past years these systems have been extensively considered, and various programming

[1] This work has been partially supported by a grant of Italian MURST and British Council (Investigation of Fault Tolerant Mechanism for Parallel Systems in Real Time Applications).

methodologies have been proposed to design and implement them. These programming methodologies take into account the fact that reactive systems are concurrent and fault tolerant programs. Fault tolerance is achieved through the use of some form of fault tolerant action which is intended to enhance the system dependability. Special techniques like software diversity (Avizienis and Kelly, August 1984) are used if a very high level of reliability is required.

The number of different programming methodologies proposed to implement dependable reactive systems is quite high, and in some cases the decision about what is the best choice may be a difficult one due to these sometimes contradictory possibilities. What is missing, in our opinion, is a strategy to define in a clear way the requirements of the applications so that the appropriate programming methodology can be selected.

In this paper we summarize different programming methodologies using two main paradigms. They have been named living processes and hidden concurrency are distinguished by their approach to concurrency structuring. The characteristics of each paradigm are discussed in Section 2. Then we outline the structure of a general multi-component architecture for reactive systems. The use of application requirements and application constraints checklists is introduced as a way to select the appropriate implementation paradigm. Finally, we show that it is possible to define an intermediate paradigm that combines the characteristics of living processes and hidden concurrency and can be used in a wide range of applications.

2. PROGRAMMING PARADIGMS FOR REACTIVE SYSTEMS

The general structure of a reactive system is represented in Figure 1. It is possible to identify three main system components:

- the system state vector;
- the set of events;
- the control software.

The system state vector provides a representation of the controlled physical system; the set of events defines the possible changes in the state of the controlled system, and the control software implements the governing mechanism of the system which reacts to events and keeps the system state in the desired conditions.

To implement this type of system it is necessary to define the relationships among the different components as well as the internal organization of each single component. Many issues have to be taken in consideration at this regard like the decision

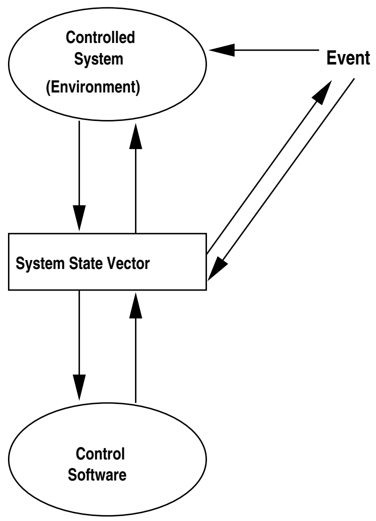

Fig. 1. A general representation of a reactive system

of using a centralized or distributed system state vector and the respective control access rules, the consequent organization of the control software and other.

To increase the dependability of concurrent systems, several approaches for software fault tolerance have been proposed; among them we recall the Conversation (Randell, 1975) and the Concurrent Recovery Block (Kim, 1982). Both papers propose to design the system as a set of distributed atomic actions, the most critical of them can have alternative implementations, called "alternates". The primary alternate is executed first and the acceptance test is checked. If it is not ensured then the next alternate is tried. This continues until either the test is ensured or all alternates fail (in the latter case the action signals failure).

2.1 *Living Processes Approach*

A program can be composed of a set of processes, each one assigned to perform one specific kind of work (such as to control part of a plant). Groups of processes cooperate from time to time, in order to reach a common goal. We will refer to this solution as living processes paradigm. A more dependable cooperation can be developed inside using the Conversation scheme (Randell, 1975). A conversation is entered by a set of processes as reaction to the occurrence of an event. Alternates are defined inside each process (see Figure 2).

Several implementations for fault tolerant actions have been defined, see for example (Clematis and Gianuzzi, 1991),(Gregory and Knight, 1985), (Tyrrell, 1987).

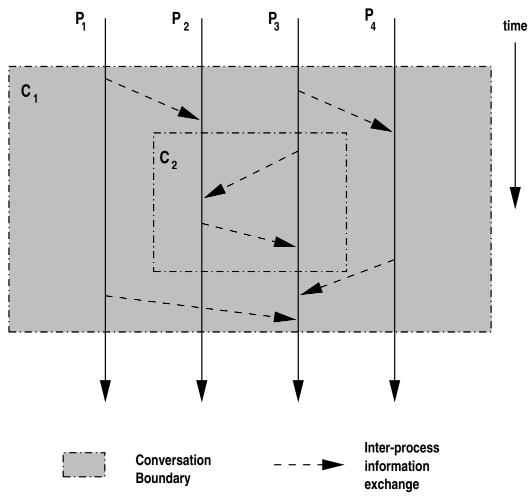

Fig. 2. Processes P1, P2, P3, P4 take part in conversation C1; only P2 and P3 are involved in nested conversation C2.

In (Tyrrell, 1987), a centralized coordinator is used for each conversation to provide fault tolerant control of a distributed systems, written in Occam. A conversation is entered by its constituent processes and a conversation control process acts as a test line coordinator for the conversation. When a conversation is started, a nominated member of the set of entry processes initializes the conversation coordinator, which exists for the duration of the conversation.

In the approach proposed in (Clematis and Gianuzzi, 1991), described using Ada, the coordinator of the conversation is a monitor which lives forever, since the same conversation could be executed more than once. In both proposals, each process implements its conversation code as a set of alternates.

The third example is the Colloquy, defined using as an extension of Ada (Gregory and Knight, 1985). Here each colloquy (a conversation-like structure) is a separated program section to which processes enrol in order to participate. Each process executes part of this code. Different processes can enter each alternate. In this case, static analysis of the program is not feasible, thus a deadlock can occur.

In all these proposals we have the following restrictions:

(1) Shared data composing the state vector are recorded within processes they are managed by. They are statically distributed.
(2) Depending on the event, processes decide to enter the conversation in which they are interested.
(3) Alternates are defined inside each process.
(4) Hardware faults can be tolerated restarting the involved processes on different nodes, from their saved checkpoints.

All the papers cited above examine in deep the problems related to the implementation of the conversations, such as deserter processes, information smuggling, deadlock among conversations, but they are lacking in considering problems related to the overall system design, such as: which is the scheme to be chosen in a given application, which part of the system is responsible for verifying the event occurrence and for activating the related conversation, how can the state vector be distributed.

2.2 *Hidden-Concurrency Approach: Event-Driven Atomic Actions*

There are several concurrency models for which this cooperation scheme may be not the best solution. A complementary model is the hidden concurrency paradigm, in which each action is implemented by a set of processes the live of which is limited to the temporal extension of the action. Typically, such a scheme is like a master/slave approach: a control manager selects an action, and the workers perform the selected task. Such a model could be more useful when parallelism is used for performance improvement, for example, in operating systems for a parallel machine providing fault tolerant services.

In such a paradigm, alternates are defined by different sets of concurrent processes (Kim, 1982), (Romanovsky and Strigini, 1995). For example, paper (Romanovsky and Strigini, 1995) offers a practical method for using backward recovery and software diversity within Ada. The authors consider conversations for coordinated backward recovery of concurrent processes and propose:

- a restricted scheme similar to Kim's "concurrent recovery block" (Kim, 1982), but providing for deadlines on the execution of the diverse modules;
- programming rules for applying this scheme to Ada procedures; and
- a way for automatically enforcing these rules through a source code pre-processor.

Within this scheme each conversation is packaged as a fault-tolerant (ft-) procedure which has several diverse designed bodies (alternates) - see Figure 3. They are tried sequentially one-by-one. The state vector is managed by the starter process, and variables are passed to the processes implementing the action as IN-OUT parameters. The execution of each body consists of concurrent execution of several tasks which are forked when the variant starts and jointed when it is finished.

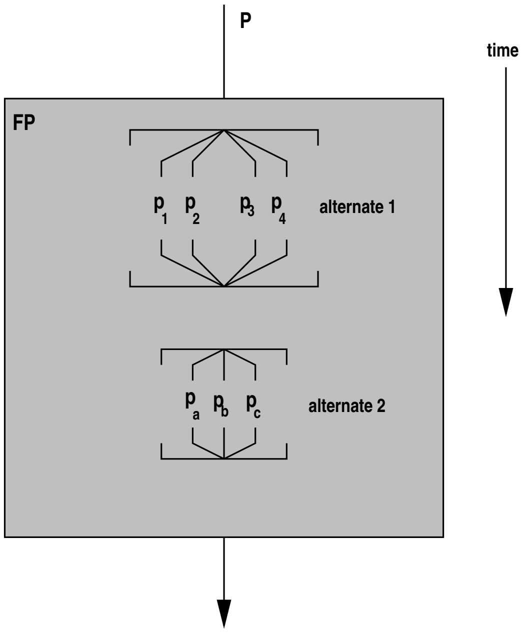

Fig. 3. The execution structure of a ft-procedure FT: its alternate has four internal tasks (p1, p2, p3, p4) and the second alternate has three tasks (pa, pb, pc). Process P calls this procedure and when the execution of alternate1 fails, alternate2 is executed.

The number of tasks may be different in different alternates, depending on the algorithm chosen in the alternate for implementing the ft-procedure. In addition, each ft-procedure has an acceptance test which should be checked when an alternate has been completed.

To conclude, within the ft-procedure scheme the concurrency is hidden inside alternates and the user of the ft-procedure is unaware of this, as well as he/she is unaware of software errors which may happen during the ft-procedure execution.

The two paradigms discussed above are complementary: the static distribution of processes and of the state vector avoids the problem of continuously activating remote processes and of distributing the state vector, while the distribution of the action bodies allows to take advantage of the centralized state vector management and makes it easier to avoid information smuggling and deadlock.

3. APPLICATION REQUIREMENTS AND SYSTEM CONSTRAINT ANALYSIS

Considering the model of a reactive system presented above it is evident that these systems may be designed using a multi-component architecture and a stepwise refinement methodology. The relationships among the components elements or modules can be specified in a implementation independent way, and then the implementation may be carried out using the different programming structures discussed in the previous section. The choice of a specific implementation strategy depends on different factors and mainly:

- the type of application and its requirements;
- the selected hardware architectures;
- the type of fault tolerance that the system should provide.

As discussed in Section 2 at the higher level of abstraction the three main component elements in this architecture are: the system state vector; the set of events; the control software.

At a successive level of abstraction further information about each component and their relationships have to be defined:

- the system state vector is a set of atomic data (e.g. the value of a sensor).
- events affect the system state, and are detected by the control software looking at the system state.
- the control software detects events by checking predicates on the system state, when an event is detected the appropriate control action is triggered.

Up to this point the system may be described independently of the implementation strategy, which is then selected considering application requirements and possibly constraints like use of distributed or centralized hardware architecture.

Different general requirements can be considered:

- the physical topology of the application: this permits to understand if the system tends to be centralized or distributed;
- the type of interaction between the controlled system and the external world: typically a system is interfaced with the external world through sensors and actuators. In many cases a console exists which permit to monitor and command the system itself; in other cases we could be in presence of a multi-terminal system like an automatic banking station;
- the degree of coupling and mutual relationships among the system sub-components: in most cases this is quite low and the interactions among more than two sub-systems is quite limited.

Dependability issues will regard at least the following items:

- Is the continuity of the system required?
- Do the functions of the system require the use of complicated algorithm, so that algorithmic diversity is necessary?

System constraints may concern both hardware and software and may drive towards a specific paradigm. For example the use of a centralized or distributed hardware platform has often to be intended as a system constraint. Another system constraint is the indication of an implementation programming language.

4. A PRACTICAL APPROACH TO DESIGNING COMPLEX DEPENDABLE SYSTEMS

Reactive systems for real time distributed applications are usually written following the live process scheme. Consistent parts of these processes perform local control and only occasionally a joint action is required, for example when the operator or an alarm situation impose a change in the working parameters. Moreover, sensors and actuators are usually positioned in distributed way; thus, it makes sense to manage the state vector values locally for each process.

Living processes approach looks to be the natural choice in order to add software fault tolerance. However, such a scheme exhibits limited possibilities in process interaction and, since the alternate are defined inside each process, offers few possibility for implementing diversity.

Hidden concurrency is more suitable: the unit of replication is the whole action, thus different teams could write the actions using possibly different languages, number of processes and synchronization paradigms. If different languages are used for each alternate, they are likely to be more independent, as experiments have indicated. However, this scheme could be not efficient, for example for the need of creating processes and terminating them each time an action is activated, even when locally executed, for the need of maintaining state vector values on a central process, which could act as bottleneck.

An intermediate scheme, hereafter called the **hierarchical** scheme, which collects suitable features of both, can however be studied. More than a completely new scheme, it is a different implementation of hidden concurrency: the user gives a high level description of the distributed program following such a scheme, giving a list of actions together with their alternates. The code is then distributed among already existing processes, following possibly automatic transformations of the original program.

The final aspect of these systems is that of the living processes scheme, however it is reached by means of successive refinements starting from the action set. The starting point is a set of fault tolerant distributed actions, activated by one or more centralized processes, and working on a set of state variables, as we have shown before.

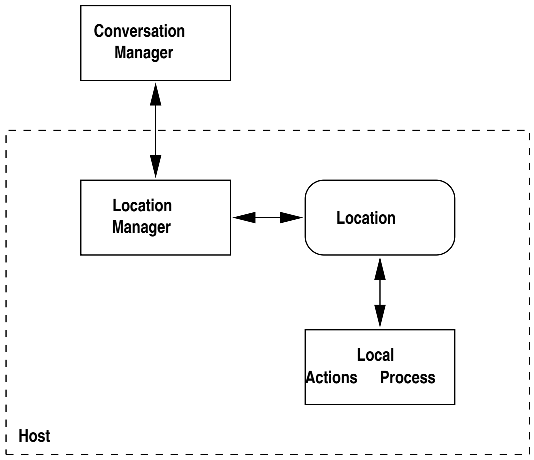

Fig. 4. Hierarchical approach structure

Let us now describe the steps to be performed to achieve the final representation.

A new logical entity is defined: the *location*, that is a subset of state vector variables. The distribution specification of the target hardware system results in the localization of the variables on the physical location to which are connected the sensors/actuators related to those variables. A logical location corresponds to each such physical locations. In this way, a partition of the state vector variables is performed.

For each location a coordinator (*location manager* is defined. This process manages the location variables, activates the *local actions* passing them the needed parameters, and synchronizes with the central coordinator to notify changes in its location variables. Moreover, a set of processes is also defined: each one is composed by a set of *local actions*, that is sequential code which is part of a fault tolerant action. Each process executes code written using a given language and communication paradigm. The central coordinator maintains a copy of the state vector, continuously updated by the distributed coordinators. It decides the activation of actions requiring the interaction among different locations, ensuring concurrency consistency (see Figure 4).

Given that scheme, the first step is to define the location variable sets for each node. Then, for each action alternate, one ore more code components are separated, each one related to a physical location. The alternate sequential components allocated on the same node are inserted in a single process. A component is activated on request of the location manager. Checkpoint and rollback actions are performed locally.

With respect to the hidden concurrency scheme, the continuous activation of new processes is avoided, assuming that an existing process is suspended at the end of an action and resumed when another its component must be activated. The local coordinator allows a more efficient work in local operation, when no distributed coordination is needed, and the locality of the useful state vector prevents from an unnecessary communication with the central coordinator. Moreover, different languages and communication styles can be used in implementing action alternates.

5. AN INDUSTRIAL APPLICATION EXAMPLE

We consider an industrial system for plastic material pressing. The system is composed by n production lines, that is n pipelines of machines, and two belts, one to feed the lines with raw materials and another one to carry out the finished goods (see Figure 5).

The problem is the environment control in the plant with respect to the following requirements: speed maintenance with respect to a given speed for each line, pressing quality, loading and drawing belts speed coordination. We refer henceforth to this requirement as the target. Each time the state changes, either for a machine failure or upon operator request, the system must be reconfigured, to conform itself to the new state. The set of working machineries and their operating level (full or partial) is referred as *system configuration*. The operator is allowed to redefine the target of the system and to know the actual system configuration.

The target is obtained controlling the machines by means of a distributed computer system: each machine and each belt is controlled by a host. The running software must either maintain the machine operativity, following the local target, and cooperate with the other computers in order to reach the global target.

A high availability of the system is required, and a strict respect of operating directives is needed to avoid damages to manufacturing machineries. Thus, some level of hardware and software redundancy must be provided in the system to avoid machineries stressing and to be able to maintain the required conditions in spite of machineries going temporarily out of order.

Besides local events that must be handled with locally executed actions, a subset of events and actions can be pointed out, requiring the interaction among different machineries, such as the following ones:

```
E1. the operator sets the target
        Each line is initialized separately
        Speed values of the two belts are
        automatically evaluated.
E2. machine alarm
        The line to which the machine
        belongs is slackened or stopped.
        Belt speed is modified accordingly.
        Operator is alarmed.
E3. belt alarm
        Possible stop.
        The machines and the other belt
        must be slackened.
E4. machine (or belt) restoring
        Re evaluation of the optimal speeds.
        Operator is informed.
```

Each action modifies the system configuration, a collection of data which can be:
- accessed in order to know the occurrence of an event,
- modified during an action,
- and known locally to control the machine or belt.

Sensors and actuators are connected locally on each host.

5.1 *Implementation with living processes approach*

The system shown above is usually described in terms of a collection of distributed processes, each one carrying out a sequential piece of code and keeping local system state up to date. To implement the conversation scheme, processes must synchronize for reaching an agreement on which event is occurred, thus the system behavior is also described in terms of possible process interactions.

The scheme of each process is:

```
loop forever
   collect local data;
   ..................
   when global_event(i) then
       local_Action(i); endif;
   ..................
   when time=Dtime then
       local_Action_timeout; endif;
   ..................
end loop;
```

where $global\_event_i$ is a predicate evaluated on variables maintained by remote processes, and $local\_Action_i$ is a local sequential code. Alternates are defined inside each local action like in Recovery Block scheme, thus limiting the diversity; deadlock is possible, since processes could independently decide to enter different actions, and the general execution flow of the distributed software system can be not easily understandable, since processes can interact in unexpected ways.

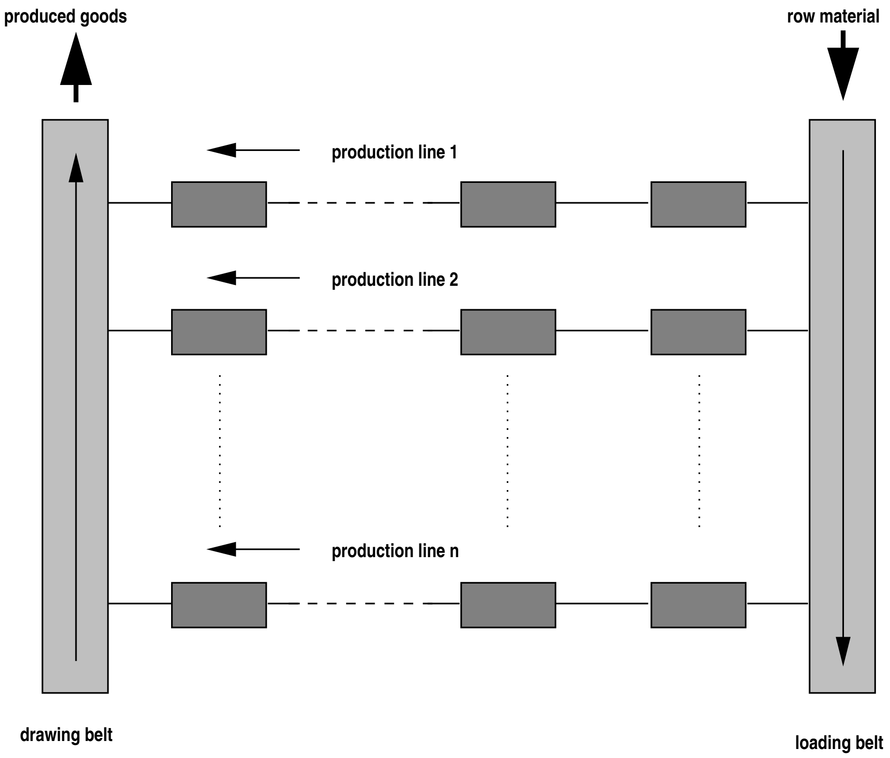

Fig. 5. Industrial system for plastic material pressing

5.2 *Implementation with hidden-concurrency approach*

In this case, system configuration is kept by a centralized process, which continuously monitors the system (thus sensor data must be sent to it from decentralized hosts) and activates remote actions.

The scheme of such process is:

```
loop forever
   collect remote data;
   ....................
   when event(i) then
              Action(i);
   ....................
   when time=Dtime then
              Action_timeout(j);
   ....................
end loop;
```

where $event_i$ is a predicate evaluated locally on the system state. Following the approach proposed in (Romanovsky and Strigini, 1995), the action body of a critical action is composed by a set of alternates. Each alternate should include the list of hidden processes, the locations in which they must be activated, their code and a list of IN and OUT parameters (part of the system state). Remote processes are created/destroyed each time, wasting computing time and possibly loosing real time deadlines. However, this approach allows the programmer to clearly define triggered atomic actions, to easily avoid information smuggling and deadlock, and possibly to assign priority among actions.

In the previous example, for each event, the occurrence of which is checked by the central process, a distributed action must be defined. Its execution is triggered by the central process itself, which also activates remote processes.

5.3 *Implementation with hierarchical approach*

In the hierarchical approach the most favorable features of both approaches can be exploited.

At the higher level there is the definition of the state vector variables and of a set of triggered conversations, as for the hidden-concurrency approach, in order to describe the system behavior in terms of interaction units. The higher level description follows:

```
State Vector Variables:
           ..........  /* Data */
   Conversation set:
     {
      ......................
      Conversation Ci:
      when event_i(Ev(Ci)) do
        if alternate_1
           do Action_1(Data(Ci_1))
        if alternate_2
           do Action_2(Data(Ci_2))
```

```
     .............
     if alternate-n
       do Action_n(Data(Ci_n))
    ensuring acc_test_i(ATData(Ci))
  .....................
}
```

where $alternate_j$ is a global variable initially set to 1 and implemented each time the acceptance test fails, $Action_j$ is a distributed atomic action and $Ev(C_i)$, $Data(C_{i,j})$, $ATData(C_i)$ are the subsets of the State Vector Variables involved in the computation.

At the lower level, the description of the processes implementing each conversation action is given.

State Vector Variables can usually be partitioned in logical units (called *locations* in section 4), each one loaded on the host to which it belongs. Each location is also bounded to the set of processes (composing the conversation actions and defined at the lower hierarchical level) working on those variables. Thus, for each location, a unique process can be built, composed by a set of sequential atomic actions, triggered by the occurrence of a global event. Event occurrence can be checked by a centralized process, or by using a distributed handshaking, as proposed, for example, in (Back and Kurki-Suonio, 1988).

6. EPILOGUE

We have considered two different approaches to the design and implementation of fault tolerant reactive systems. We have pointed out that through application requirements and constraints analysis it is possible to identify the characteristics of a suitable programming paradigm. A new programming paradigm has been derived in this way, and its applications to a test case study has been discussed.

With respect to two paradigms which we have identified in the paper, these of the living processes and of the hidden concurrency, the hierarchical approach maintains a high readability and allows an easy testability or correctness checking as the second approach, together with the efficiency and reality adherence of the first approach.

The logical description of the system starts with the state vector definition, the description of the events and of the related actions. Each action is replicated in several alternates, each one implementing a possibly concurrent different algorithm.

By means of the variable localization, that is of their partitioning on the target hosts, it is possible to distribute the actions as well. Then, by means of a process of stepwise refinement, the single sequential alternates are merged in a unique process, but keeping them separately, so that each one can be activated on request of the location manager.

The local manager is then the local control process. Moreover, since reliability is better achieved if the different versions are developed using different programming languages, each action alternate can be written using a different language and assuming a particular communication and synchronisation style. Thus the location manager, which manages the location (the local part of the state vector) must present a specialized interface for each version.

The location manager is responsible for coordinate:
⋆ the hosts, since it communicates to the central coordinator state vector updates and receives from it the local action activation request,
⋆ the local action process, to which it sends the activation request,
⋆ the different local action versions, since it also manages the location variables, independently from the programming paradigm implemented by each version.

7. REFERENCES

Avizienis, A. and J.P.J. Kelly (August 1984). Fault tolerance by design diversity: concepts and experiments. *IEEE Computer* pp. 67–80.

Back, R.J. and R. Kurki-Suonio (1988). Distributed cooperation with action systems. *ACM Trans. Progr. Lang. & Sys.* **10(4)**, 513–554.

Clematis, A. and V. Gianuzzi (1991). Software fault tolerance in concurrent ada programs. *Microprocessing and Microprogramming* **32, 1–5**, 365–372.

Gregory, S.T. and J.C. Knight (1985). A new linguistic approach to backward error recovery. *Proc. 15th Symp. on Fault Tolerant Computer Systems* pp. 404–409.

Kim, K.H. (1982). Approaches to mechanization of the conversation scheme based on monitors. *IEEE Trans. on Soft. Eng.* **SE-8(3)**, 189–197.

Randell, R. (1975). System structure for software fault tolerance. *IEEE Trans. on Soft. Eng.* **SE-1(1)**, 220–232.

Romanovsky, A. and L. Strigini (1995). Backward error recovery via conversations in ada. *Software Engineering Journal* **10(8)**, 219–232.

Tyrrell, A.M. (1987). Increasing software reliability of distributed systems with occam. *Proc. 2nd Int. Conf. Computers and Applications* **10(8)**, 219–232.