Functional Paradigm for Designing Dependable Large-Scale Parallel Computing Systems

Andrea Bondavalli* and Luca Simoncini**

* CNUCE-CNR, Via S. Maria, 36 - 56126 Pisa - Italy, E-mail: andrea@mammolo.cnuce.cnr.it
** Dip. di Ingegneria dell'Informazione, Università di Pisa, Italy, E-mail: simon@iet.unipi.it

Abstract

This paper proposes the use of a functional language and of a dataflow computing model for the design of large-scale parallel computing systems for which dependability, in its aspects of reliability, timeliness, parallelism and distributedness are requirements of main concern. The proposed design methodology is sufficiently flexible to allow for verification and validation of such systems from the very first steps of the design. The advantages of such an approach resides in the potential for parallelism it admits, in being characterised by referential transparency and in the property of composability. The design description language is extended for dealing with dependability issues, with the creation of a library of fault tolerance schemes which can be used for modular insertion of redundancy. A set of tools are being defined as part of a design development environment allowing the designer to proceed in successive interactive steps, each of which can be validated.

1. Introduction

The design of dependable large-scale parallel computing systems is very complex for many reasons; among these we focus our attention on the complexity of typical applications, to which size is often a contributing factor, and on the need to take into account requirements, like dependability and distribution, which are often called non-functional, as they cannot be described in terms of a functional relationship between inputs and outputs, although they obviously contribute to define the intended behaviour of the system. For the first point a largely used approach to designing systems is to distinguish between in-the-large design [6], [11] and in-the-small design. An in-the-large design has the purpose of describing, by using a proper language, called design description language, the overall structure of a system without unnecessary details of the individual modules. The in-the-large design lacks the details internal to each module, which are described in the in-the-small design, but contains all the interface specifications, allowing the necessary consistency checks. For taking into account requirements like dependability and distribution, which are crucial for the intended behaviour of the system and are to be verified as early as possible in the design process for being confident in a given level of predictability of the design, the in-the-large design should consider, and consequently the design description language should have the possibility to describe, such properties both for design and verification purposes [5].

The programming paradigm of the design description language which is used for the in-the-large design may be relevant in mastering the added complexity which is due to the size of the system and to the description of non-functional properties and may help in their early verification. In this context, we discuss the advantages of design description languages for the in-the-large design based on a functional paradigm and a dataflow computing model for the design of dependable large-scale parallel systems, and the special adaptations that may be needed to represent and verify properties in designs described with such languages. We present also an outline of a suitable design development environment. This paper describes activities which are part of an ongoing project aimed at providing designers with system development environments populated by a set of tools for aiding their job. Several of these tools are under development and are not described here in detail. A similar approach which tries to combine the functional paradigm with object orientation is presented in [14].

2. Proposed model

The functional paradigm for the description of programs together with a dataflow approach for taking into account decomposition and parallelism may be the basis of a design description language satisfying many of the challenging requirements of predictably dependable large-scale parallel systems. As discussed in [7] functional languages have several interesting properties which range from the programmability, intended as the possibility of expressing programs solving complex problems in an elegant and concise form, to potential for parallelism, intended as the possibility of implementing programs in a
highly parallel fashion, due to the lack of side effects and the absence of imperative flow of control, to inherent fault tolerance, due to the referential transparency property by which it is possible to tolerate simple failures by re-evaluating the same function on the same input data which produces always the same result [3], [7].

In the dataflow model a system is described as a directed graph (where the nodes are the modules and the arcs are communication links). A dataflow program may be represented by a graph, whose nodes correspond to modules processing streams of data flowing in through the input arcs, and producing a stream of results flowing out from the output arcs. The nodes, or modules, are executed asynchronously on the basis of their execution (firing) rule solely. They interact with the rest of the system only through the operations of input, with activation and execution of the module, and output of computed results with termination of the module itself. Dataflow languages are able to express all the potential parallelism inherent to the functional programming approach. They have also the very interesting advantage of maintaining the functional behaviour of the composing modules and the referential transparency property inherited from the functional approach. Since modules composing a system have functional behaviour and locality of effects, dataflow languages have the property of composability which puts in direct relation the general behaviour of a system from its constituent parts [8]. Systems structured in this way present inherent fault tolerance in the sense that simple failures may be tolerated by simply resubmitting the same inputs to a module describing the same function [7];

From the point of view of verification and validation, the composability property implies that modules which are individually correct, and properly connected, will produce a correct system: it is impossible for a module to affect the behaviour of other modules through means other than explicit messages. Structural models for software reliability assessment can be applied since all data necessary to their use can be obtained by a simple instrumentation of software code [12], and the data-flow graph representing the system can easily be translated, with proper techniques, into some sort of Petri net [9] and then used for timeliness analysis purposes. Of course, this feature must be supported by appropriate tools populating an interactive design environment. Interaction is used for calibrating the design so that dependability and timeliness requirements can be attained with a high level of predictability.

We only consider static dataflow graphs: no modules can be created or destroyed at run-time. This has obvious advantages in the verification of a design. However, while keeping static graphs, one should consider different degrees of dynamicity in execution. In fact, there may be operations that the system has to perform only upon the occurrence of specified events, both external, e.g. timing events, and internal, e.g. fault occurrences. When a designer has to specify firing conditions in terms of the occurrence of many possible events, a useful instrument is a so-called non-deterministic construct, which lets the implementation free to choose one of them at random. On the other hand, the presence of non deterministic constructs leads to programs for which the referential transparency property does not hold.

Our choice is not to admit non-deterministic constructs, but to include the possibility of conditional execution of modules. The firing rule is modified allowing the execution of some module based on the presence of a specified subset of input messages. In addition conditional execution may be constrained by conditions on message values. Last the possible conditions specifying the firing of a module have to be disjoint either by mutually exclusive predicates or conditions or by assigning different, may be dynamically changeable, priorities.

3. In-the-large design with a dataflow notation

An important issue to address is the subdivision of design information between in-the-large and in-the-small parts. A reasonable and useful compromise is to develop a 'basic' in-the-large language with a basic set of constructs by which the components are defined with their interfaces and linked together, enriched with comment notations by which additional properties or characteristics of the entities in the design can be described.

An important concern in choosing a design notation is top-down development via recursive decomposition (detailing) of modules. This is possible with dataflow models: a module in the design is substituted with a dataflow graph, which becomes a subgraph of the system design. Some consistency rules between representations of a design at different levels of decomposition (detail) are straightforward. However, there are properties that may be desirable in dataflow modules and are not preserved through composition of modules. Specifically: i) a dataflow graph made of state-less modules is not state-less if it contains cycles; ii) a dataflow graph made up of 'atomic' modules (by which we mean that their execution is atomic with respect to communication with the outside, and possibly with failures) is not itself an 'atomic' module. It may sometimes be appropriate to set a rule that state-lessness and/or atomicity must be preserved through decomposition steps.
3.1. An in-the-large language

The primitive entities of a design are modules, which are deterministic and may have internal state, and links. Modules may first be defined as types (or classes) and then be used: instances of them may become components of other modules. A fundamental part in the definition of the modules is the definition of their interfaces. An interface is a set of ports through which the entity will interact with the external environment. The definition of ports is part of modules definition and later, when instances of modules are used, the ports are linked together according to safe and correct disciplines. All ports have a name, local to the module, and may be more or less strictly typed (strict typing seems generally advisable: the format of the messages is pre-determined).

A module may be a simple module, i.e.: a basic entity realised in the in-the-small language, or a composed entity: a subgraph. Simple modules may be further subdivided into modules for which only one firing rule is defined and those expressing conditional execution for which several alternative firings are defined. The former are executed only when data are available on all input ports and produce results on all the output ones. Each firing in the latter is expressed by means of a guard and by the subset of output ports on which the results will flow. The guard is composed by the subset of input ports on which data must be available, a predicate defined on this data and on the internal status of the module and a priority for enforcing the mutual exclusion among firings. The firings are described in the comment associated to the module.

A graphical syntax may be defined by representing modules as rectangular boxes and ports as triangles, input ports as intersecting the box representing an entity with one vertex inside and two outside and output ports with one vertex outside and two inside. The presence of a state in a module is represented by a shaped area in the right part of a box. The basic definition of simple modules and an example are in Fig. 1.

Subgraphs are composed of boxes which may in turn be either subgraphs or simple modules. Seen as 'black boxes' they are defined in the same way as the simple modules, i.e. with their names and interfaces; but then the boxes are 'opened' and their definition as 'white boxes' must include also the types and instances of their components and the linking of components ports and the ports of the subgraph itself. The definition of a subgraph and an example are in Fig.2.

An important issue in determining the correctness of a design is the consistency of the connections among the entities. It is straightforward to define conditions for correct connections:

• a connection is between two and only two ports;
• it must start from an output port and end in an input port;
• the types of the connected ports must be compatible (the simplest rule of compatibility is that the two ports must have the same type);
• connections cannot cut entity boundaries.

A design is then described by defining its graph, i.e., listing the nodes and their input and output ports, linking source and destination nodes, and assigning types to each module, that is, the name of the in-the-small software entity or of the subgraph with which it must be implemented.

Comments are used in the description of a module and/or subgraph to explicitly indicate the semantics of the module/subgraph, timing and dependability requirements derived from the specification, and timing and dependability constraints related to the peculiar module/subgraph. Comments are used in the top/down design as requirements for the decomposition of the system; they are a guideline for the identification of a consistent set of modules/subgraphs, each with its related comments for the decomposed system. Each phase of the top/down decomposition can be validated by proving the consistency between the components interconnected as specified in the design together with their comments and the comments of the system which has been decomposed.
4. Dependability issues in system design

Dependability issues account for both redundancy management, for inserting fault tolerant techniques in the system, and timing issues, for dealing with timeliness constraints if the system has to deliver real-time services. The benefits of dealing with dependability issues in the in-the-large design has been previously motivated; in the following we examine the specific problems.

4.1. In-the-large description of redundancy schemes

Structuring redundant graphs requires the definition of special modules and subgraphs, with associated comments, which have to take into account and solve the problems associated with the management of redundancy [4].

Any software redundancy scheme is composed of a set of variants which are available for being executed, either sequentially or in parallel, depending on a given fault tolerance strategy and whose results must be validated by a properly chosen adjudication policy. If the variants are executed sequentially and the flow of computation can be controlled by a successful validation of the results of each variant we get an implementation of the recovery block scheme [13]; if the variants are executed in parallel and all the results contribute to the input of the adjudication function we get an implementation of the N-version programming scheme [1]; both parallel and sequential execution of variants are the basis for the self-checking programs scheme [10] and finally if some sort of optimisation of available resources is part of the strategy we have the SCOP scheme [2]. The implementation of any software redundancy scheme needs the explicit definition, and insertion in the design of the scheme, of modules which support:

1) the correct ordering of different instances of the specific software redundancy scheme;
2) the adjudication functions;
3) the sequential execution of the variants and the setting of recovery points;
4) the parallel execution of the variants.

Ordering of instances. The structuring induced by using the proposed dataflow notation implies the automatic exploitation of pipe lining. This poses the problem of supporting ordered executions of different instances of a computation and particularly of instances of the same software redundancy scheme. If a subgraph has a single flow of data towards the output ports, ordering of activations is maintained through the entire execution of the subgraph itself; if it has more than one flow of data towards the output ports, it may happen that results pertaining to different activations are scrambled and the correct ordering must be restored. In the in-the-large description, this may be taken into account by defining a proper module as the last module in the subgraph. This module, of type Collect, has to collect the results produced from different activations of the scheme and to release them in the correct order. It has as many input ports as the number of output ports of variants which may participate to different instances of the same scheme, and it has an output port for releasing results and an optional another one for signalling failures of the entire subgraph. The associated comment includes as many firings as the number of different exit points, with predicates which account for failure of variants in different activations of the scheme.

Adjudication functions. The in-the-large description of any software redundancy scheme requires the declaration of special modules, of type Check, which have the function of adjudicating redundancy scheme results produced by the variants used. The number of input ports depends on the particular scheme in which a module of type Check is used; the output ports in general should be connected to the output port of the subgraph describing the chosen scheme. The function representing the behaviour, defined in the comment associated to the module of type Check will depend on the chosen scheme; in a recovery block it will be an acceptance test on the results produced by the considered variant while in a N-version programming scheme, it will be either a classical majority voting, or whatever sort of voting is considered appropriate.

Sequential execution of variants. Let us consider a software redundancy scheme, like a recovery block, made of a set of variants which are to be executed sequentially; the results from each variant has to be validated by the adjudication function; let consider moreover that all the variants must be fired on the same set of input data. The capabilities which are needed for allowing sequential execution and setting of recovery points are: 1) choosing a single firing set for the variants, and 2) saving the input data to be used for the firing of successive variant (on failure of the previous one).

1) a special module, of type Activ, can be defined. Its input ports will have to be connected to the input ports of the subgraph implementing the recovery block; of its two output ports, one is to be connected to the input port of the first variant and one to another special module of type Prop (see later); the associated comment will contain all the possible firings.

2) a set of modules, of type Prop, can be defined. There is one of such modules for each variant; it has to propagate to the next variant the firing set chosen in case the present one has failed, or stopping propagation in case of its success. The input ports of modules of type Prop
are to be connected to the output port of the module describing the adjudication function and to the previous module of type Prop. The associated comment specifies two firings, which correspond to the failure or success of the associated variant. The textual and graphical declarations of a subgraph implementing a recovery block with two variants is shown in Fig.3.

```
SUB Recovery_Block;
IN data:type1;
OUT res:type1; failure: Boolean;
USE Activ, Variant, Check, Prop, Collect, Exh;
ITEMS MA:Activ; M1, M2:Variant; MC1, MC2:Check; MP1, MP2:Prop;
MColl: Collect; MEH:Exh;
CONNECT  data TO MA.a, MA.x TO M1.a, MA.y TO MP1.a,
         M1.x TO MC1.a, MC1.x TO MP1.b, MC1.y TO MColl.a,
         MP1.x TO M2.a, MP1.y TO MP2.a, M2.x TO MC2.a,
         MC2.x TO MP2.b, MC2.y TO MColl.b, MP2.x TO MEH.a,
         MP2.y TO MEH.b, MEH.x TO failure, MColl.x TO res;
```

**Fig. 3. The subgraph representing a recovery block scheme**

**Parallel execution of variants.** Let us consider a software redundancy scheme made of a set of variants which are to be executed in parallel, like in a N-version scheme; each variant must contribute with its results to the adjudication function which has to select the result of the whole construct. All the variants must be fired on the same set of input data; the capabilities which are needed for allowing parallel execution are: 1) choosing a single firing set for the variants and starting the parallel execution of all of them; 2) preventing state divergence among correct variants. This can be obtained by declaring a module, of type Distr, slightly different from the type Activ defined previously. Its functionality is the gathering of the necessary data inputs and their distribution to the variants. The differences with respect to Activ is the number of output ports which must be equal to the number of input ports of all the variants. Its associated comment contains all possible firing of the variants. The textual and graphical declarations of a subgraph implementing a 3-version programming scheme is shown in Fig.4.

```
SUB 3_Version;
IN data:type1;
OUT res:type1; failure: Boolean;
USE Distr, Variant, Check;
ITEMS MD:Distr; M1, M2, M3:Variant; MC1:Check;
CONNECT  data TO MD.a, MD.x TO M1.a, MD.y TO M2.a, MD.z TO M3.a,
         M1.x TO MC1.a, M2.x TO MC1.b, M3.x TO MC1.c,
         MC1.x TO res, MC1.y TO failure;
```

**Fig. 4. The subgraph representing a 3-versions scheme**

The special modules described in this paragraph are used for defining a set of basic subgraphs which describe a library of redundancy schemes which can be invoked by the designer in structuring the system, depending on the fault tolerance requirements which have to be fulfilled by the design. The comment associated to each of this basic subgraph contains several fields which have to be filled out by the designer; relevant to redundancy management are 1) the number of variants used in the scheme, and 2) the reliability figure which the designer wants to obtain. This latter information can be used in the top-down refinement phase for identifying information on the reliability figures which must be attained by the used variants, while in the validation phase the same information can be used for verifying that the figure has been attained by composing, with known rules, the reliability figures of the modules composing the scheme.

**4.2. In-the-large description of timing properties**

In order to take into account timing issues in the in-the-large design i) modules/subgraphs and communication arcs must be labelled with execution times (as deterministic bounds or probability distributions) and ii) the dataflow diagram has to be directly translatable into some sort of timed Petri net, so that it can be analysed not only for specific liveliness and safety properties, but for timing properties as well. There are different ways for extending dataflow languages with provisions for dealing with timing issues. Leaving the application modules completely unaware of time, and labelling modules with the required activation times, is a solution which appears
simple, in that it allows a complete separation of timing concerns from all other design issues, but overly restrictive, in that it disallows the use of timing information by the application programmer (for instance to decide at run time on the handling of time-out situations). Our choice is to provide in a design two different kind of timing information. The first is used for writing applications that explicitly must take into account timing information or to perform occasional operations, that are not to be repeated at each computation frame, or for expressing time dependent conditions for raising exceptions to react to missed deadlines. The other information is concerned with the requirements to impose to the underlying machine. They are used for expressing the time-related characteristics of the modules of the design, such as execution time, which, in turn, must be used for allocation and scheduling purposes.

The approach we adopt is a declarative one by which the information related to the application are expressed as supplementary condition on the firings of a module and those related to the time characteristics and attributes are expressed in the comment associated to each module. Its advantage is that most modules can be programmed and verified without taking into account timing issues. At the same time, all the timing requirements are made explicit in the high-level design instead of hidden in separate scheduling requirements.

The problem remains of determining the execution times of individual modules. However, the possible sequences of execution of the modules are directly described in the dataflow graph, and the comments make available all the other necessary information for planning a static scheduling of module execution (or for additional inputs for dynamic scheduling, like priorities among modules). In addition, using the information supplied at the application level and assuming satisfied the requirements imposed on the underlying support, a design can be proved to satisfy its timing specifications, stated either in terms of relative time or absolute time.

5. Programming tools and environment

Designing a large system requires a careful use of a set of tools which helps the designer to proceed in a methodical and controlled way, and which can be used to assess the quality of the design since the very first phases. Instead of having a fully integrated programming environment, we prefer to provide the designer with a set of tools which can be invoked independently one from the other depending on the special requirements the designer has to cope with. The proposed environment supports operations on different stages of the design of an application. It includes (many of these operations will be invoked in a feed-back loop of design-verification-redesign):

- the description, by the designer, of the basic dataflow graph of the in-the-large design and the assignment of module types: a graphical editor for the dataflow description part of the in-the-large language is thus needed as well as a library of in-the-small modules; the described graph may still contain subgraphs; some checks, like type matching on ports connection, can be performed at this stage;
- the top down refinement of the design, by decomposition of the subgraphs, so to get a description at a higher level of detail;
- the automatic verification of liveness, safety, etc., properties of the functional in the large design, using a Petri net model;
- the specification by a designer of global timing requirements using the appropriate comment language;
- the verification of global timing requirements on the fully labelled design,
- the specification, by the designer, to transform the graph by assigning reliability figures to modules and subgraphs; in this phase the designer has at his/her disposal a library of redundancy schemes which are used to extend the design with the insertion of required fault tolerance,
- the automatic verification, at this level of detail, of the fault tolerance requirements, by using a tool for the automatic production of Markov models for reliability analysis of the system
- compilation with the in-the-small description of (all or some) in-the-large modules in the functional in-the-large design to obtain an executable program;
- the description, by a designer, of the hardware of the target system, in terms of available processors, memory, and whatever else is needed for allocating and scheduling tasks;
- the labelling of the modules in the in-the-large functional design with processor allocation labels;
- the labelling of the modules in the in-the-large functional design with execution time labels;
- schedulability check on the fully labelled in-the-large design using the hardware description and knowledge about the execution environment; and/or the derivation of a static schedule or of information (e.g. priorities) for a dynamic scheduler;

5.1. Outline of the graphical editor

The graphical editor, which has been developed, allows the designer to draw the basic components of the dataflow graph, by choosing between boxes of two kinds: modules and subgraphs, and characterising the existence or not of internal state for each of them. The modules name
correspond to the assigned type of the modules, and the input and output ports have to be typed as well. The modules are then added to the design database for use in different part of the design or for being used as basis for the generation of other modules. The designer can then connect with arcs each output port from a module/subgraph to the input ports of other modules (including the same). The editor controls the type matching between the two ends of each arc, forbidding the drawing of an arc between not matching ports. The designer can zoom in and out on subgraphs to get the desired level of view. Zooming is used also for further description of the inside structure of a subgraphs. On request of the designer, the editor opens a window for each module/subgraph for allowing the designer to comment it. Comments as just said contain information on the function the module/subgraph has to perform, its initial state (if needed), reliability requirements, number of versions, kind of redundancy scheme, timing information, as well as descriptive comments for documentation. Compilation of the description produces an extended graph with all subgraphs exploded at the level of detail allowed by the used libraries. In the case of the subgraphs associated to redundancy scheme, the comments associated to the composing modules are partially filled out consistently with the reliability labelling of the subgraph and the chosen redundancy scheme and number of versions.

Two other tools, which are under development, can be invoked to verify the consistency of the reliability labelling, the liveness properties and the general timing properties of the design. At any stage, if something is not satisfying for the designer, the program graph can be interactively changed and all the process repeated since the desired level of consistency with requirements is obtained.

6. Conclusions

This paper proposes to approach the design of dependable large-scale parallel computing systems by using a functional language as design description language and a dataflow computing model. This allows to master the complexity related to the system itself and for taking into account non-functional attributes like dependability, in both aspects of reliability and timeliness, distributedness and parallelism. It seems particularly fitted for systems in which dependability, fault tolerance and distributedness are requirements of main concern. Fundamental issues like the subdivision of design information between in-the-small and the in-the-large design and the choice of the comment notations to be used to support design and verification have been addressed and a basic design description language is used for the definition of the structure of the system and then extended for dealing with dependability. Other issues like the exact definition of the comments information and of the tools that would process these comments in order to verify and validate a predictable behaviour of the system are object of our current work. All these tools are the basis of a design development environment which allows the designer to proceed in successive interactive steps, each of which can be validated.

References