A NEW COMPRESSION TECHNIQUE FOR TOOLS THAT USE DATA-FLOW GRAPHS TO MODEL DISTRIBUTED REAL-TIME APPLICATIONS

Stergios Papadimitriou (1, 2), Achilles Kameas (1),
Panayiotis Fitsilis (1, 3) & George Pavlides (1, 2)

(1) University of Patras
Department of Computer Engineering & Informatics
Patras, Greece

(2) Computer Technology Institute
Patras, Greece

(3) Intrasoft
Athens, Greece

Abstract *: This paper deals with the effective representation of data flow graph (DFG) models for distributed real-time applications. To this end, we first present the real-time data flow graph (RDFG) model, which is a modification of the traditional DFG model that we will be using to cope with the special requirements of distributed real-time tasks. Then we present a new technique for RDFG compression, which can also be applied to DFGs. Using it, CASE tools based on the DFG model can benefit both in storage and speed by transforming the graph to a more compact form. The algorithm consists of three phases applied consecutively to a DFG that has been previously transformed to a task-tree. The whole process has been developed in the context of SEPDS project, which aims at producing an experimental CASE tool based on the DFG model. Finally, we present the conclusions drawn from the analysis of the computational complexity of this technique, together with future directions of our research on the subject.*

Keywords *: CASE tools, software development environments, real-time applications, data flow graph*

1 Introduction

Data Flow Graph (DFG) models have been introduced in an effort to overcome the lack of granularity of analysis inherent to the traditional task modeling approaches that used the *task* (defined as the smallest processing entity that is dispatchable by an operating system) as a modeling unit [3,4,5]. Although methodologies that use DFGs to represent the task structure [2,11,10] offer increased granularity of analysis, they also lead to large analytical and simulation models which often require enormous computing resources. The application of a compression process can alleviate this problem by constructing an equivalent and much smaller DFG.

In this paper, we first present (section 2) the Real-time Data Flow Graph model (RDFG), which is a modification of the traditional DFG model we have made in order to represent effectively the execution of real-time workloads. This model has been successfully incorporated in SEPDS [7,8], an experimental Software Development Environment for distributed applications based on DFG models. Then we present and analyze a new technique for DFG compression (section 3). Our compression algorithm transforms a DFG into an equivalent one which is much smaller. Thus CASE tools based on the DFG model can benefit both in storage and speed by transforming the DFG to a compressed form.

Our improvements over previous works on compression processes [1] are threefold. First, we introduce two new constructs to the traditional DFG in order to represent the parallelism at the fine and coarse grained levels and also study the compression rules of the new constructs. Second, the proposed task tree based compression process is simpler and more efficient. Finally, our modification to the traditional DFG results in a model which integrates elegantly the Task Flow Graph (TFG) [10,1] and the Generalized Stochastic Petri Nets (GSPNs) [1,6]. The designer has one simple and flexible model to become familiar with (the modified DFG) and the computational overhead associated with the transformation between two models (TFG and GSPN) is avoided.

In the end, we present results concerning the complexity and the validity of the presented compression technique, as well as the state of our current research on the subject.

2 The Computational Model

The traditional DFG model, as it was presented in [7,8] forms the basis of our computational model. In order to make it a suitable basis for the *decomposition* and *performance evaluation* of large distributed systems, we have made a number of extensions.

We have associated an execution time, $\delta(a)$, with each actor. It can depend both on the specific data values that the actor uses and on its execution state (described by the state links presented below). In addition, the $PRE(a)$ and $POST(a)$ functions are generalized; now they may be arbitrary logic functions over the IFS and OFS, respectively.

A new link type, the *state links* is introduced, in order to capture the state in which an actor operates at each time instant. Finally, a new field, $TYPE(a)$ is introduced as an indicator of whether an actor is primitive, or can be further decomposed. It can be used to describe the system at different levels of detail.

Consequently, an actor a is defined as a 5-tuple:

$$a = (PRE(a), POST(a), FUN(a), \delta(a), TYPE(a))$$

The variation of the DFG model presented above is called the EDFG model. Depending on the input links that affect the execution of an actor (called the Input Firing Set - IFS) and the output links affected by this execution (called the Output Firing Set - OFS), we can define a set of *Primitive Graphs* (PGs), together with the associated execution rules; this set in the EDFG model contains four PGs [8]: *And Primitive Graph (APG)*, *Or Primitive Graph (OPG)*, *Select Primitive Graph (SPG)* and *Replicate Primitive Graph (RPG)*.

The EDFG model is, however, inappropriate for systems having strict timing constraints, because it is too general a model to fullfill the need for formal analysis and validation that these systems have. To facilitate the representation and analysis of real-time processes, we have added to the above set a number of PGs.

The *Or Primitive Graph Loop (OPGL)*, *Select Primitive Graph Loop (SPGL)* are used as entry (OPGL) and exit (SPGL) points of a loop; branching back probabilities and branching dependent delay factors can be associated with SPGL. In addition, the *Not Primitive Graph (NPG)*, which operates much like a hardware inverter, and serves a similar purpose with the *inhibitor arcs* of extended Petri Net models [6].

Also, *two chain primitive graphs, CHain Start(CHS) and CHain End(CHE)*, which simply pass their inputs to their outputs are used to represent the ordering of the actors in a chain sequence. A *Replicate Parallel Primitive Graph (RPPG)* serves to model the spreading of a task on many processors in a multi-processor environment. When it fires, it activates all the subgraphs that are connected to its output links. Each such subgraph is executed on a separate processor, while the overhead to spread the task (δ_{fork}) is accounted.

Finally, the *And Parallel Primitive Graph (APPG)* and *Or Parallel Primitive Graph (OPPG)* are used to join the execution of a parallel task previously spreaded with the RPPG, and correspond to a join point of a parallel task: an APPG to a point where all the joined subtasks must complete their execution whereas for the OPPG it suffices the completion of one subtask. The overhead to join a parallel task execution (δ_{join}) can be associated with the APPG and OPPG primitive graphs.

A real-time process needs to communicate with other processes and with its environment (via the input/output of control information) in order to accomplish its control mission. This can be accomplished by using a few *communication primitives* [1] with well defined semantics. The points of process execution where communication primitive calls are made, are called *communication points*. Each communication point can cause a rescheduling operation; therefore *to analyze the timing behavior of a task, the whole task system must be considered.* Further communication imposes precedence relations between tasks that complicate scheduling issues (parts of lower priority tasks must be completed to enable a higher priority). At the communication points *communication actors* are placed in the graph notation that correspond to the desired primitive call.

An actor that is not a communication actor, performs some computation and will be referred to as a *computation actor* or simply *actor*. The notation A_i will be used to represent a computation actor while C_i will be used for a communication actor.

Furthermore, we must apply two restrictions on this model. First, we require that all the actor firing times $\delta(a)$ must be exponentially distributed R.Vs. This is a restriction applied on the power of the model in order to make formal analysis methods based on the derivation of Continuous Time Markov Chains (CTMCs) applicable. Second, algorithms that reduce the number of actors by merging them whenever possible cannot be applied if their transformation rules are not expressed precisely. For this reason, each actor is restricted to have *one input and one output link*. Any $PRE(a)$, $POST(a)$ function is expressed in an AND/OR/NOT logic using the PGs. Also to each PG acting as an upper-end point of a subgraph, *a lower-end point PG with the same number of inputs as the outputs of the upper-end must correspond.* This is a restriction on the structure of the model that does not affect its power. Note that in order to achieve such symmetry in model structure, we must ignore the internal structure of the communication actors, which depends on the communication semantics and is completely irrelevant to this work, since we regard communication actors only as boundaries on the processing of other components.

The DFG model with these modifications made to address the special requirements of real-time systems and especially of the analytical methods is referred to as the **Real-Time Data Flow Graph** (RDFG) model.

3 The compression process

An RDFG can be decomposed into six types of subgraphs depending on the execution semantics imposed on the components of those subgraphs. A component of a subgraph is either an actor or recursively a subgraph. The components are recursive elements that can be replaced by any of the six subgraphs or by an actor (computation or communication). When the components of a subgraph form independent paths in the graph notation they will also referred to as its *branches*. The six types of subgraphs and their execution semantics are (the PGs that serve as subgraph boundaries are also given in parentheses):

- the **Chain-Subgraph - CSG** (CHS to CHE): *All* its components are executed serially *in the order in which they appear* (which imposes precedence constraints between them).
- the **And-Subgraph - ASG** (RPG to APG): *All* its branches must be executed before control flows out but they can be executed *in any order*. Each branch must have at least one actor.

- the **Or-Subgraph - OSG** (SPG to OPG): *Only one* branch is executed. A probability density function determines the branch selection. There can exist one branch with no actor (NULL branch).

- the **Parallel-Min - PMINSG** (RPPG to OPPG): Each of its branches is executed on a separate processor and the *first branch* that completes is sufficient for the execution of the whole subgraph.

- the **Parallel-Max - PMAXSG** (RPPG to APPG): Differs from PMINSG in that it requires the completion of *all of its branches* (each on a separate processor) in order to complete its execution.

- the **Loop-Subgraph - LSG** (OPGL to SPGL): It is introduced to allow the timing of *loop* constructs. It consists of a loop body (subgraph between OPGL-SPGL primitive graphs) that is executed $\frac{1}{1-P_b}$, $P_b \neq 1$, times where P_b is the branching back probability.

Before the compression technique can be applied, the RDFG must be transformed into a task tree. The root of the task tree is the whole RDFG (viewed as subgraph). Intermediate non-leaf nodes consist of the six types of RDFG subgraphs and the leaves are the RDFG actors. All subgraphs in the same level in the task tree are assigned the same level number. By convention, level number 0 (level 0 of the task tree) is given to the root, 1 to the subgraphs positioned directly under the root (level 1 of the task tree) and so on.

For a task tree with N levels (with the root having level number 0 and the leaves $N-1$) the compression algorithm performs $N-1$ iterations of a three-phase compression process applied *consecutively and sequentially* on the components of every level starting from the $N-2$ (second outermost) level and proceeding upwards to the root (level 0). The first phase tries to compress the subgraph on which it is applied by combining its actors (*Actor Combination Phase*). The second phase moves an actor to a subgraph of the same level if it is possible to combine that actor on every branch of the subgraph in order to reduce the total number of actors (*Actor-to-Subgraph Movement Phase*). Finally the last phase merges adjacent PGs (whenever a possible), allowing combination of actors that were initially in different subgraphs (*PG Merging Phase*).

The following three subsections describe how each phase operates on the currently active (working) level, while the compression algorithm (which simply calls bottom-up, with respect to the task tree, each of the three phases $N-1$ times) is presented at the end of the section in algorithmic format, together with the results of its analysis.

3.1 Actor Combination Phase

Actors in the same level can be combined if and only if the combination is a *correctness preserving transformation*, that is, if the precedence constraints imposed between any two actors are retained by the combination (functional correctness), and if the timing behavior of the task system is not affected (temporal correctness). This means that for all tasks, the probability densities of the times until their communication points remain unaltered (note that only the timing in respect with the communication points if of our concern).

As far as the RDFG model is concerned, the precedence constraints are retained if the combination merges a set of adjacent computation actors in a CSG or LSG (possibly all their actors) or a set of ASG, OSG or PMINSG branches (possibly all the branches). The timing behavior remains unaffected only when the delay associated with the combination accounts correctly for all the delays of the actors that it replaces.

Combinations at the same level can be **horizontal**, when two or more branches (children) of an ASG, OSG or PMINSG are combined, **vertical**, when actors are vertically combined in a CSG or LSG, or **total**, when a whole subgraph is replaced by a single equivalent module. This applies to all the subgraph types (except for the PMAXSG where we will forbid any combination for reasons related to the exponential distribution).

The following proposition gives a necessary and sufficient condition for the existence of a total subgraph combination (the proof is contained in [12]):

Proposition 1 *A subgraph S has total combination if and only if it does not contain any communication actors.*

A CSG can have vertical and total combinations. If it does not contain any communication actors then, by the above proposition, the whole CSG can be replaced by a single equivalent actor. The expected execution time of an actor derived after a combination in a CSG is computed by summing up the execution times of the combined actors.

The ASG and OSG can have only horizontal and total combinations. For an OSG with n branches, let p_i be the probability of the i-th branch and δ_i its expected execution time. Then $\delta_{OSG} = \sum_{i=1}^{n} p_i \delta_i$ in the case of total OSG combination, and $\delta_{j'} = \frac{\sum_{i \in COMB} p_i \delta_i}{\sum_{i \in COMB} p_i}$, when a subset COMB of OSG branches are horizontally combined into the single equivalent module j'. As far as the ASG is concerned, execution times are computed in the same way as in the CSG case.

The LSG can have vertical and total combinations. The vertical LSG combinations are similar to those of the CSG. For its total combination, let P_b ($P_b \neq 1$) be the loop's branching probability. Then the actor derived after the total loop's combination has mean value $\delta_t = \frac{\delta}{1-P_b}$, $P_b \neq 1$, where δ is the mean value of a single loop's iteration.

Although the expected execution times after combinations in a CSG, ASG, OSG and LSG can be obtained easily from by the semantics of the combinations, the same is not true for the PMINSG and PMAXSG. In the PMINSG (PMAXSG) cases the execution time of a combination is a random variable defined as the minimum (maximum) of the random variables that describe the execution times of the combined branches. This definition is based on the execution semantics of those subgraphs.

Thus, for the PMINSG subgraph, only horizontal and total combinations are allowed. Using the fact that the minimum of exponentially distributed RVs is also exponential [9] (since $\delta_{P_{MIN}} = \min\{\delta_1, \ldots, \delta_n\}$) we have ($\delta_{tc_s}$ and δ_{tc_j} are the overheads to spread and join the task's execution): $\delta_{P_{MIN}} = \frac{1}{\frac{1}{\delta_1} + \frac{1}{\delta_2} + \ldots + \frac{1}{\delta_n}} + \delta_{tc_s} + \delta_{tc_j}$, for the total PMINSG combination and $\delta_{P_{MIN}} = (\sum_{i \in COMB} \frac{1}{\delta_i})^{-1}$, when a subset $COMB$ of the PMINSG's actors can be horizontally combined.

The combination of actors inside the PMAXSG is not allowed, since the maximum of exponentially distributed RVs is not exponential.

The following proposition validates the combination rules which work by summing the execution times of the combined modules by proving that any error will be on a strictly conservative basis:

Proposition 2 *The compression process will produce error on a* **strictly conservative side** *(the original model has variability less than or equal to that of the compressed).*

Proofs of the correctness of the combination rules and of the above proposition are given in [12].

During the actor combination phase, if a PG is found adjacent to a communication actor, it is marked with a *stopping point.* Such a point improves the efficiency of the compression algorithm by stopping early the attempt of the next phase to move and combine actors inside the subgraph. While working with the task tree representation two flags for each subgraph node are needed. They correspond to the two stopping points needed in the RDFG for upper-end and lower-end PGs. The total and horizontal ASG combinations are illustrated in Figure 1 both in their RDFG and task tree form. Note the removal of PGs (subgraph nodes for task tree) after total combinations; it is done to prepare the processing of the next phase.

3.2 Actor-to-Subgraph Movement Phase

While the rules of the combination phase work with actors at the same level, the actor-to-subgraph movement phase contains rules that are stated in terms of an actor and a subgraph *at the same level* in the task tree representation. The general rule to follow in this phase is:

An actor is moved inside a subgraph (of the same level) if it is possible to be combined within that subgraph with a correctness preserving transformation.

Again, this rule applies differently to each subgraph type. If an outer actor A_o is adjacent to the CHS point and there is an inner actor A_i directly below the CHS point, the outer actor A_o can be moved inside the CSG and combined with A_i (Figure 2.a). The case where an outer actor exists adjacent to the CHE point and there is a CSG's actor directly above it is similar.

For the LSG, a similar operation as in the CSG case is being carried out, but the execution time of the moved actor is adjusted by multiplying it by $1 - p_b$.

If an actor is above the SPG point of an OSG and there exists an actor at each branch of it adjacent to the SPG, then the outer actor can be moved in and combined on *every* OSG branch (Figure 2.b). The case with an actor below the OSG is similar.

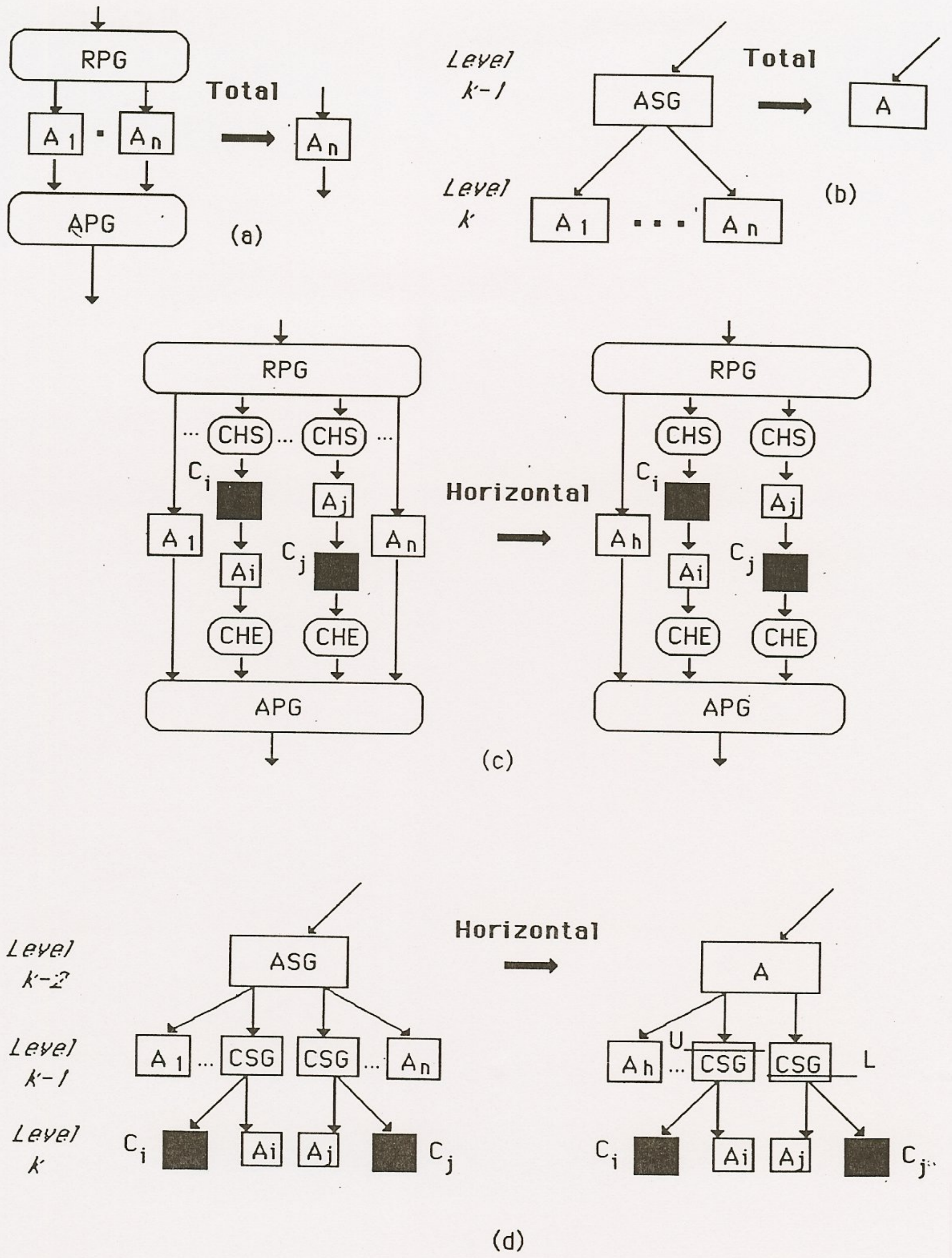

Figure 1: The total (a. and b.) and horizontal (c. and d.) And-Subgraph combinations in RDFG and task tree representations.

In these cases the total number of actors is reduced by one. When an actor adjacent to an ASG boundary (RPG or APG) exists, no combination can be done, since it would violate precedence constraints and execution time equivalence. For the same reasons, the movement and combination of actors at PMINSG and PMAXSG boundaries is also prohibited.

3.3 Primitive Graph & Subgraph Merging Rules

Some of the PGs that were adjacent in the original RDFG and some PGs that have become adjacent during the compression process are redundant and must be merged to prepare the graph for the next combination phase that will be applied to the immediate upper level. This phase operates both on PGs at the RDFG and on the subgraph nodes at the task tree representation.

Each PG merging operation raises the level of the inner subgraph by one, making possible the combination of the actor of the inner level (derived by the combination phase) with the actors of the upper level subgraph during the next combination phase, or the movement and combination of an upper level actor inside the subgraph during the next actor-to-subgraph movement phase.

Two PGs connected by an edge (adjacent) can either have the same level number or level numbers that differ by one. Thus if $G_1 \rightarrow G_2$ means "an edge exists from PG G_1 to G_2", then there exist three possible cases (*n is the working level*): G_1, G_2 are in the same level(n), G_2 is one level(n) inside the level of $G_1(n-1)$, or G_1 is one level(n) inside the level of $G_2(n-1)$. For each of the above three cases, all the possible adjacent PGs combinations must be handled with a combination rule based on the PGs executional semantics.

Examples of each of the three cases, as they appear in graphs of the RDFG model, are depicted in Figure 3. It is noteworthy to observe that the stopping points are always assigned at the outer level (smallest level number) PG. This permits the second phase to test efficiently whether or not an actor can be moved inside an OR-Subgraph. Below the rules for the cases (a), (b), (c), (d), (e) of Figure 3 will be explained. All the possible subgraph merging rules are given in [12].

In the notation we will be using to describe the PG combination rules, $G_1 \rightarrow G_2 : C \rightarrow G_1(G_2)$ means "combine the G_1, G_2 PGs and replace them with their combination $G_1(G_2)$ without assigning any stopping point", $G_1 \rightarrow G_2 : S \rightarrow G_1(G_2)$ denotes the assignment of a stopping point to $G_1(G_2)$ with no combination, and $G_1 \rightarrow G_2 : NULL$ means no action is taken.

In case (a) of Figure 3, the combination rule: $APG \rightarrow SPG : NULL$ means that no combination is possible and no stopping point assignment is done.

In case (b), the RPG forces all its branches to execute in any order but the SPG causes only one of its branches to execute. Any RPG, SPG combination will change the program's semantics and its execution time and thus it must be prohibited. The stopping point assigned to RPG by the PG combination rule: $RPG \rightarrow SPG : S \rightarrow RPG$ prevents any combination between actors separated by the RPG (even though the algorithm does not make any attempt to move inside ASG's boundaries).

In case (c), using the same reasoning, the stopping point is assigned to the OPG with the rule: $APG \rightarrow OPG : S \rightarrow OPG$ Actually the assignment of a stopping point in this case is needed to improve the performance by preventing early any attempt for movement and combination in the second phase. In Figure 3(c), the stopping point prevents early the movement of an actor inside an Or-Subgraph from its lower-end point at the $n-1$ actor-to-subgraph movement phase; in Figure 3(f) the same hold for the OSG upper-end point.

In Figure 3(d), PG merging can be performed successfully, since when control comes to the outer RPG all its n branches and the m branches of the inner RPG are fired immediately. Apparently, this is equivalent to the firing of all the $m+n-1$ branches by the RPG derived from the merging. The used rule is : $RPGouter \rightarrow RPGinner : C \rightarrow RPGouter$

In Figure 3(e) however, all the n branches of the RPG are executed serially, while the m branches of the RPPG are executed in parallel. Clearly, no PG combination can be done and RPG is marked with a stopping point: $RPG \rightarrow RPPG : S \rightarrow RPG$

When the compression process is applied to the task tree the *subgraph merging rules* are equivalent to the RDFG PG merging rules. These rules closely resemble the operation of PG merging rules. Note however, that one subgraph merging corresponds to two PG mergings (the upper-end and the lower-end PG mergings). As a consequence, the rules for subgraph merging are simpler and more effective that their PG counterparts.

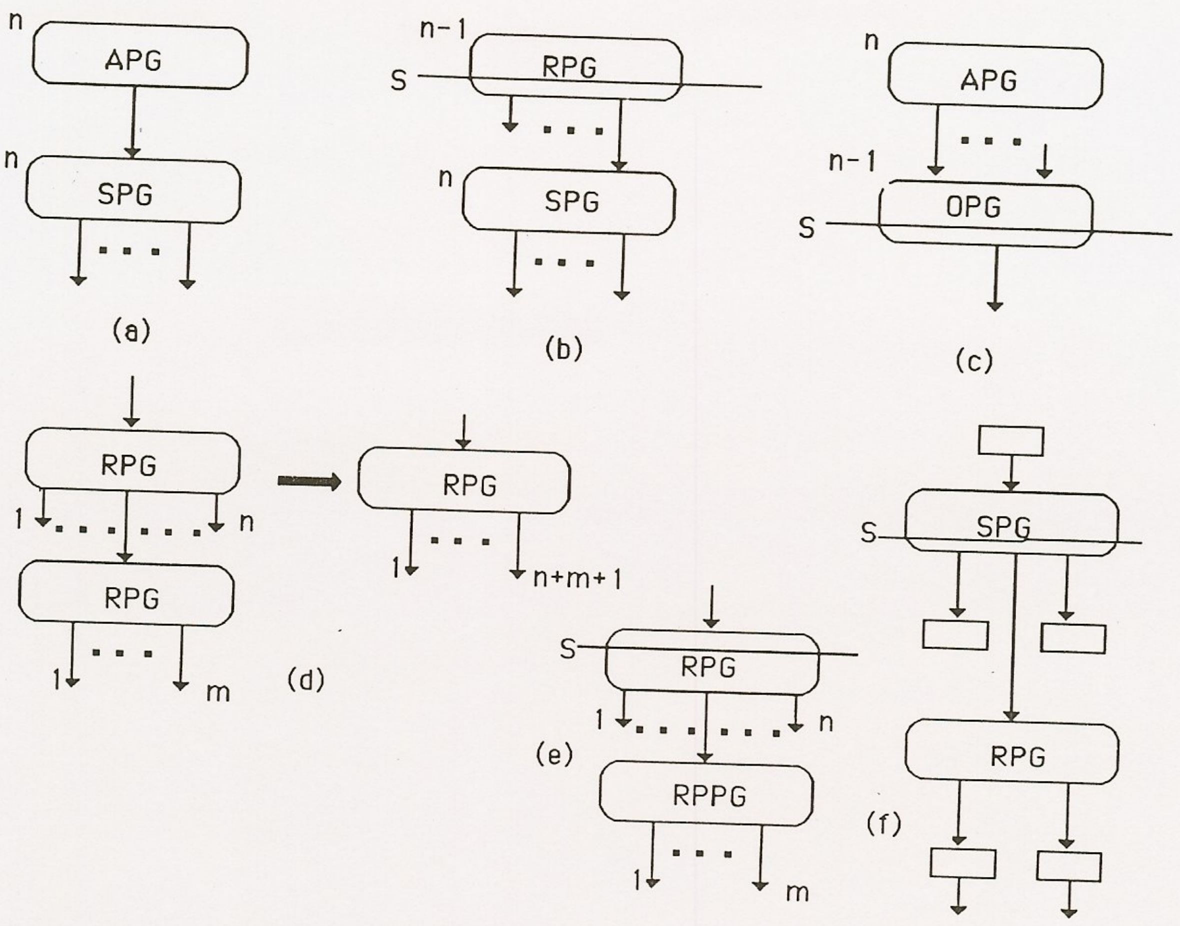

Figure 2: Illustrating some actor-to-subgraph movement & combination rules.

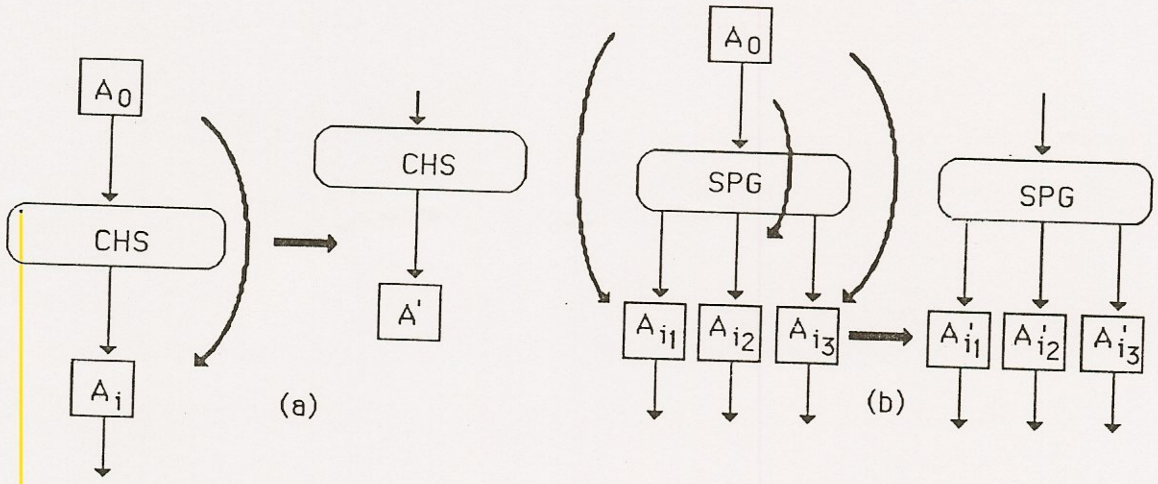

Figure 3: Illustrating some PG merging cases

3.4 The Compression Technique and the RDFG compression algorithm

In order to compress a DFG, the *sequential* and *repetitive* application of all of the three phases previously described is needed. Consequently, starting from the second outermost level, we proceed towards the root of the task tree representation of the DFG, as we apply consecutively the three phases.

The compression algorithms that result from the application of the compression technique to the RDFG model (similar algorithms can be easily derived for other DFG models), for both the RDFG and task-tree representations (0 to $N-1$ are the N RDFG levels) are:

The RDFG compression algorithm	The task tree compression algorithm
Input: *an RDFG and its leveling information*	*a task tree*
Output: *the compressed RDFG*	*the compressed task tree*

```
The RDFG compression algorithm
for working_level=$N-2$ downto 0 do
begin
actor_combination(working_level);
actor_to_subgraph(working_level);
PG_merging(working_level);
end;
```

```
The task tree compression algorithm
for working_level = $N-2$ downto 0 do
begin
leaf_combination(working_level);
leaf_to_subgraph(working_level);
subgraph_merging(working_level);
end;
```

The expected cost of the compression process is $O(N_c)$ (see [12]), where N_c is the number of nodes of the original (uncompressed) task tree. Since the required processing time increases linearly with the number of nodes, large task trees (and consequently, large DFGs) can be processed efficiently. By studying the amount of achievable compression (using a Markovian model as the basis of the study), we have found that it mainly depends on the percentage of computation actors. By keeping the other involved parameters (that is, the relative frequencies of occurrence of each type of subgraph) constant while varying this percentage, we have obtained results that back up the expected fact: *as the percentage of computation actors increases, the degree of achievable compression increases also (with a fast rate as well).* The data in the following table (which were obtained by solving numerically the Markovian model) give a quantitative view of this result (note that 20 % of the components are subgraphs and 80 % are actors):

Percentage of computation actors	50 %	55 %	60 %	65 %	70 %	75 %	80 %
Degree of compression	0.551	0.617	0.686	0.759	0.836	0.916	1.000

Note that in the last line of the above table, all the graph becomes a single actor, since no communication actors are contained in it. A complete complexity analysis of the presented algorithm can be found in [12].

4 Conclusions

The aims of this paper were twofold: first, to present a suitable data-flow graph model (RDFG) for the representation of real-time distributed workloads, and second, to give a new, more efficient compression technique that can be applied to all DFG-based models.

The application of this technique to the RDFG model has been analytically presented and analyzed. In brief, in the context of this technique, the RDFG is first transformed to a task-tree which is a more compacted and suitable for further analysis representation, after a leveling algorithm has been applied to it in order to organize its components by level number. Then a three-phase compression algorithm is used to compress the task tree (or the RDFG) and to create the equivalent one with the smallest possible number of leaves (executable modules).

In a future work, we will generalize the compression process in order to handle actors with execution times which have probability distributions different from the exponential, as well as execution times which do not follow a single probability distribution.

The compression process presented here is the first phase of an integrated technique for the analysis of real–time distributed applications. The other two phases (which will be also presented in future works) of this technique are *a reachability information computation and a CTMC model construction phase*, during which the reachability information is computed for the nodes of the compacted task tree, and the CTMC model of the system for the first stable cycle (initial CTMC) is constructed, and *a CTMC model analysis phase*, where the time evolution of the initial CTMC, in the presence of time-driven task invocations and of asynchronous events, gives rise to several CTMCs, which describe formally the task system. The applications of the dynamic programming algorithm and of real-time probabilistic temporal logics in the context of those CTMCs are important research topics that we are currently studying.

This technique, together with the RDFG model, have been incorporated in SEPDS [7,8] a DFG-based SDE for distributed applications. Some modifications to the original design of SEPDS have been made, in order to make it appropriate for the specification and analysis of real-time distributed applications. We are currently in the state of testing the proper functionality of the system and hope to be able to present results on the system efficiency soon.

Acknowledgments

The authors wish to thank Prof. Paul Spirakis for his helpful comments on earlier drafts of this report, and Dr. Anry Levy for many fruitful discussions on early stages of the research.

References

[1] D. Peng, and K.G. Shin, "Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control", *IEEE Transactions on Computers*, **C-36(4)**, Apr 1987, pp 500-516.

[2] A.K. Mok, P. Amerasinghe, M. Chen, S. Sutanthavibul and K. Tantisirivat, "Synthesis of a Real-Time Message Processing System with Data-driven Timing constraints", *Proc. IEEE Real-Time Systems Symposium*, Dec 1987, pp 133-143.

[3] K. Ramamritham, J.A. Stankovic and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements", *IEEE Transactions on Computers*, **38(8)**, Aug 1989, pp 1110-1123.

[4] J.A. Stankovic, K. Ramamritham and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems", *IEEE Transactions on Computers*, **C-34(12)**, Dec 1985, pp 1130-1143.

[5] J.P. Huang,"Modeling of software partition for distributed real-time applications", *IEEE Transactions on Software Engineering*, **SE-11**, Oct 1985, pp 1113-1126.

[6] M. Ajmone Marsan, G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems*, MIT Press, 1988.

[7] A. Levy and G. Pavlides, "Simulation vs. Prototype Execution: A Case Study", *Proc EUROCOM-91.*

[8] A. Levy, J.van Katwijk, G. Pavlides and F. Tolsma, "SEPDS:A Support Environment for Prototyping Distributed Systems", *Proc First Int'l Conf on System Integration*, New Jersey, USA, Apr 1990.

[9] A. Papoulis, *Probability, Random Variables and Stochastic Processes*, McGraw-Hill, 1984.

[10] C.D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[11] F. Distante and V. Piuri, "Distributed architecture to match optimum process allocation: A simulated annealing based approach", December 1987, pp 114-123.

[12] S. Papadimitriou, G. Pavlides and A. Kameas, "A new compression algorithm for data-flow graph models of distributed real-time tasks", *Technical Report TR.92.04.09*, Computer Technology Institute (CTI), Patras, Greece.