J Supercomput (2008) 43: 241–255
DOI 10.1007/s11227-007-0143-3

Modeling and simulation of parallel adaptive divide-and-conquer algorithms

Fernando J. Barros

Published online: 28 June 2007

Abstract Conventional modeling and simulation formalisms only give support to the representation of model behavior, providing no constructs for describing changes in model structure. However, some systems are better modeled by self-reconfigurable formalisms. We have developed the *Discrete Flow System Specification* (DFSS) to exploit dynamic structure, component-based and hierarchical model construction. Due to structural similarity, dynamic self-configuring DFSS models offer a good description of systems, like adaptive algorithms and reconfigurable computer architectures. In this paper, we present the modeling and simulation of a parallel adaptive divide-and-conquer integration algorithm in the CAOSTALK modeling and simulation framework, a realization of the DFSS formalism.

Keywords Simulation · Dynamic structure models · Adaptive algorithms

1 Introduction

A dynamic structure model can transform itself in a new model by changing its components and/or the connections among them. A large number of systems change their own structure; examples include reconfigurable computer architectures [5, 11, 12] sensor networks [7] artificial neural networks [13] and biological systems [9], to name a few. The study of these systems has been difficult due to the lack of modeling and simulation formalisms with structural semantics that can mimic the structural changes in the real systems.

In previous research, we have developed the concept of general system dynamic structure network and its specialization for describing discrete event models [1]. The Discrete Flow System Specification formalism (DFSS) provides a component-based

F.J. Barros (✉)
Department of Informatics Engineering, University of Coimbra, 3030 Coimbra, Portugal
e-mail: barros@dei.uc.pt

approach to modeling and simulation. DFSS components can be built hierarchically and hierarchical components can change their structure dynamically.

The parallelization of numerical algorithms has been extensively studied and a large efficiency has been achieved with parallel machines. State-of-the-art parallel computer architectures involve the use of a large number of processing elements connected by a communication network that can be dynamically reconfigurable [3, 8]. Other platforms developed for massively parallel computation, like grid computation infrastructures [4], also exhibit runtime reconfiguration characteristics.

Efficient algorithms are in many cases adaptive requiring a dynamic partition of the problem so more computational effort is put into regions where a pre-specified accuracy is more difficult to obtain [6]. These algorithms perform commonly better that the corresponding non-adaptive versions that use the same effort in every part of the problem being their performance limited by the most demanding sub-problem. In function integration, for example, a non-adaptive algorithm divides the overall integration interval in equal width sub-intervals. To obtain a given accuracy the width needs to be chosen so it can provide a good approximation in the overall integration interval. An adaptive algorithm uses different widths to deal with the different segments of the integration interval so it is able choose the best width to obtain the required precision. This type of algorithms can use a large width in the regions where the integral can be easily computed and smaller widths where the integral yields to larger errors.

Simulation plays an important role in assessing the performance of many systems that cannot be adequately represented with analytical models. Simulation has been employed to study the performance of computer architectures [14] and operating systems algorithms for dynamic load balancing [15], for example. Traditional simulation methodologies provide a good representation for static structure models [14]. However, the representation of dynamic data structures used in adaptive algorithms cannot be conveniently described with these simulation methods. We consider that a simulation methodology able to mimic the dynamic nature of adaptive algorithms provides a superior representation and enables easier to develop and to maintain models.

We describe in this paper a modular representation of adaptive algorithms. Each node in the algorithm is modeled by an independent simulation component. These nodes can be created dynamically to represent the processes spawned in the underlying computer to handle the new subtasks found during algorithm execution. Transmission links need also to be created in run-time to represent the communication channels existing between a new node and the existing ones. After a task finishes processing, it can be destroyed so the computer can be more efficiently used and memory released. We consider that a good representation of these dynamic algorithms requires operators able to change the model structure in runtime.

In this paper, we present the model and the simulation of a parallel divide-and-conquer adaptive numerical integration algorithm using hierarchical and dynamic structure components. Models are described in the CAOSTALK simulation environment. The benefits of a recursive component-based approach are discussed and simulation results are presented.

2 The DFSS formalism

The Discrete Flow System Specification formalism (DFSS) is a system theory based formalism aimed to represent dynamic structure models [1]. DFSS provides a hierarchical and modular approach to modeling and simulation activities. Two types of models are supported in the formalism, namely, basic and network models. Basic models interact through a well-defined set of input and output values, enabling a software component-based orientation to model representation. These models have their definition based on the Discrete Event System Specification (DEVS) [14]. Basic models can be combined to form dynamic structure networks, providing the constructs to the representation of complex self-reconfigurable models.

2.1 Basic model

In DFSS basic models are defined by the structure [14]

$$M = (X, Y, P, \omega, s_0, \delta, \lambda),$$

where X is the set of input values, Y is the set of output values, P is the set of partial states (*p-states*), $\omega : P \to \mathbf{R}_0^+$ is the time advance function, $S = \{(p, e) | p \in P, 0 \leq e \leq \omega(p)\}$ is the state set and e is the time elapsed in the current p-state p since the last transition, $s_0 \in S$ is the initial state, $\delta : S \times X^\phi \to P$ is the transition function, with ϕ representing the absence of value $X^\phi = X \cup \{\phi\}$, $\lambda : P \to Y$ is the partial output function.

The transition function describes how a component changes its state. A component in state (p, e) will change to the state $(\delta((p, e), x), 0)$ when it receives an input x after e time units elapsed since component last transition. A component in state $(p, 0)$ receiving no input during the interval $[0, \omega(p)]$ will change autonomously to state $(\delta((p, \omega(p)), \phi), 0)$. A component at state (p, e) produces the output value $\lambda(p)$ when $e = \omega(p)$ and no value (ϕ) when $e \neq \omega(p)$.

Example (Non-buffered server model) To show the behavior of atomic models we consider the representation of a non-buffered server system. When the server is idle, an arriving job will be sent out after service time. For simplicity, we consider deterministic service times. A job arriving when the server is busy is sent immediately without being serviced.

The server can be in several phases according to its status. The set of phases is given by $\Phi = \{\text{idle, busy, reneging}\}$. The model of the server is given by

$$NBS = (X, Y, P, \omega, s_0, \delta, \lambda),$$

where
$X = \{j_1, j_2, \ldots\}$ is a set of jobs
$Y = \{(port, job) \mid port \in \{\#\text{out}, \#\text{ren}\}, job \in X\}$
$P = \{(phase, \beta, job, rt, out) | \in \Phi, \beta \in \mathbf{R}_0^+, job \in X^\phi, rt \in \mathbf{R}_0^+, out \in Y^\phi\}$
$\omega(phase, \beta, job, rt, out) = \beta$
$s_0 = ((\text{idle}, \infty, \phi, \infty, \phi), 0)$

$\delta((\text{idle}, \infty, \phi, \infty, \phi), e, j) =$
$\quad(\text{busy}, 5, x, \infty, (\#\text{out}, j))$
$\delta((\text{busy}, \beta, job, rt, out), e, \phi) =$
$\quad(\text{idle}, \infty, \phi, \infty, \phi)$
$\delta((\text{busy}, \beta, job, rt, out), \beta, j) =$ $\qquad //e = \beta$
$\quad(\text{busy}, 5, x, \infty, (\#\text{out}, j))$
$\delta((\text{busy}, \beta, job, rt, out), e, j) =$ $\qquad //e < \beta$
$\quad(\text{reneging}, 0, job, \beta - e, (\#\text{ren}, j))$
$\delta((\text{reneging}, 0, job, rt, out), e, \phi) =$ $\qquad //j = \phi$
$\quad(\text{busy}, rt, job, \infty, (\#\text{out}, job))$
$\delta((\text{reneging}, 0, job, rt, out), e, j) =$ $\qquad //j! = \phi$
$\quad(\text{reneging}, 0, job, rt, (\#\text{ren}, j))$
$\lambda(\text{busy}, \beta, job, rt, out) = out$

Server phase diagram is represented in Fig. 1. In the diagram component, autonomous transitions at the end of time β are represented by dashed arrows.

The current implementation of the formalism is made in the CAOSTALK modeling and simulation environment [2], a Smalltalk implementation of the DFSS formalism. The transition function is represented in Listing 1.

The transition $\delta(p, e, x)$ is implemented by the method **delta:x:**. Given the object orientation of Smalltalk, p-states are a property of components and they are not used in the parameter set. Equality of real numbers is checked by method **equalTo:** and the absence of value is represented by **nil**.

2.2 Network model

Network models are a combination of DFSS basic models. In contrast to conventional modeling and simulation formalisms, the structure of a DFSS network can undergo dynamic transformations, the representation of self-reconfigurable systems. The DFSS *dynamic structure network* is formally defined by [1]

$$HFN_N = (X_N, Y_N, \eta, M_\eta),$$

where N is the network name, X is the set of input values, Y is the set of output values, η is the name of the dynamic structure network executive, M_η is the model of the executive η.

The model of the executive is an extended DFSS model, defined by

$$M_\eta = \big(X_\eta, Y_\eta, P_\eta, \omega_\eta, s_{0,\eta}, \delta_\eta, \lambda_\eta, \Sigma^*, \gamma\big),$$

where Σ^* is the set of network structures, $\gamma : S_\eta \to \Sigma^*$ is the structure function.

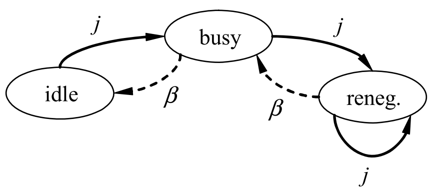


Fig. 1 Phase diagram of non-buffered server

```
delta: e x: j
   phase = #idle ifTrue: [
       phase := #busy.
      job := j.
      out := #out@j.
      ^beta := 5.0
   ].
   phase = #busy & (j isNil)
   ifTrue: [
      phase := #idle.
      ^beta := HFSS::Infinity
   ].
   phase = #busy & (e equalTo: beta)
   ifTrue: [
      job := j.
       out := #out@j.
      ^beta := 5.0
   ].
   phase = #busy ifTrue: [
       phase := #reneging.
      rTime := beta − e.
      out := #ren@j.
      ^beta := 0.0
   ].
   phase = #reneging & (j isNil)
   ifTrue: [
      phase := #busy.
      out := #out@job.
      ^beta := rTime
   ].
   phase = #reneging ifTrue:[
      out := #ren@j.
      ^beta := 0.0
   ].
```

Listing 1 Non-buffered server transition

The network structure $\Sigma_{j,e} \in \Sigma^*$, corresponding to the state $(p_{j,\eta}, e) \in S_\eta$, is given by the 4-tuple

$$\Sigma_{j,e} = \gamma(p_{j,\eta}, e) = \big(D_j, \{M_{i,j,e}\}, \{I_{i,j}\}, \{Z_{i,j,e}\}\big),$$

where D_j is the set of component names associated with the executive state $(p_{j,\eta}, e)$ for all $i \in D_j$, $M_{i,j,e}$ is the model of component i for all $i \in D_j \cup \{\eta, N\}$, $I_{i,j}$ is the sequence of components influencers of i for all $i \in D_j \cup \{\eta\}$, $Z_{i,j,e}$ is the input function of component i, $Z_{N,j,e}$ is the network output function.

For simplicity, we assume here that the models, input function and network output function do not change with executive elapsed time e and thus, $M_{i,j,e} = M_{i,j}$ and $Z_{i,j,e} = Z_{i,j}$.

These variables are subjected to the following constraints for every $s_{j,\eta} \in S_\eta$:

$$\eta \notin D_j, \qquad N \notin D_j, \qquad N \notin I_{N,j},$$

$$M_{i,j} = (X_{i,j}, Y_{i,j}, P_i, \omega_i, s_{0,i}, \delta_{i,j}, \lambda_{i,j}) \quad \text{is a basic model, for all } i \in D_j \text{, with}$$

$$\delta_{i,j}: S_i \times X_{i,j} \to P_i,$$

$$Z_{i,j}: \bigtimes_{k \in I_{i,j}} V_{k,j} \to X_{i,j}, \quad \text{for all } i \in D_j \cup \{\eta\},$$

where

$$V_{k,j} = \begin{cases} Y_{k,j} & \text{if } k \neq N, \\ X_N & \text{if } k = N. \end{cases}$$

The network output function is given by

$$Z_{N,j}: \bigtimes_{k \in I_{N,j}} Y_{k,j} \to Y_N.$$

Structural changes include modifying network composition by adding or removing components. Component interaction can also be configured in runtime by changing the set of influencers, input function and network output function.

3 Adaptive integration algorithms

Divide-and-conquer (D&C) algorithms provide efficient methods for solving numerical problems [6]. The parallel implementation of these algorithms can further increase their performance. The efficiency depends on the algorithm design, the computer architecture and the mapping of the algorithm into the computer. The architecture can be dynamic, adjusting itself to the algorithm. Performance can also be increased by the dynamic assignment of processes to computer processing elements in order to reduce communication and computation time.

The goal of this research is to study the performance of parallel adaptive divide-and-conquer algorithms used in numerical integration. These algorithms pose a new challenge to computers architecture for they require sophisticated techniques for dynamic load balancing. Algorithms can also benefit from self-reconfiguring computers able to adapt their intercommunication network in order to reduce communication time among processes [11].

D&C algorithms have a recursive nature and work by dividing problems into smaller ones that can be solved independently. After this phase, solutions are combined to yield an overall solution to the problem.

A common choice in numerical methods is to apply the integration algorithms over the whole interval and to estimate the corresponding error. If the error is too large, the interval is divided into sub-intervals and the same method is applied recursively in each of the intervals. After the partial solutions are found, they are combined to produce the integral of the original interval.

The adaptive solution exhibits better performance over fixed step algorithms, since smaller intervals can be used where the function to be integrated produces a large error and larger intervals can be used where the error is small. On fixed step algorithms, the step size is chosen for the worst case, involving many unnecessary computations.

D&C algorithms have been successfully parallelized by exploiting the ability to solve simultaneously independent sub-problems in different processing elements [6]. We consider in this paper the use of the DFSS formalism to represent the dynamic nature of the adaptive algorithms. The ideal solution would be a reconfigurable computer that can self-adapt its configuration in order to minimize inter-process communication by, for example, creating shorter paths between processes that exchange information [11]. However, we take into consideration the more widespread reconfigurable computers [3] by modeling computation times and communication delays as stochastic processes.

The simulation requirements of adaptive algorithms can be described with a simple example. A node is created with the information of the function to be integrated and the limits of the integration interval $[a, b]$. The node computes the value int_{ab}, an approximation to the integral based on some integration algorithm, like the Gaussian quadrature or Simpson's rule, for example, and computes the value $error_{ab}$, an estimate of the error. If $error_{ab} < maxError$ then the value int_{ab} is accepted as a good approximation to the function integral in the interval $[a, b]$. Otherwise, the node calculates the middle point m of the interval and creates two new nodes to compute the sub-integrals int_{am} and int_{mb}. The integral is then given by $int_{ab} = int_{am} + int_{mb}$.

A typical behavior of D&C adaptive algorithm is described in Fig. 2, where the computation over the interval [0, 8] is described. A first node is created to handle the overall interval as depicted in Fig. 2a.

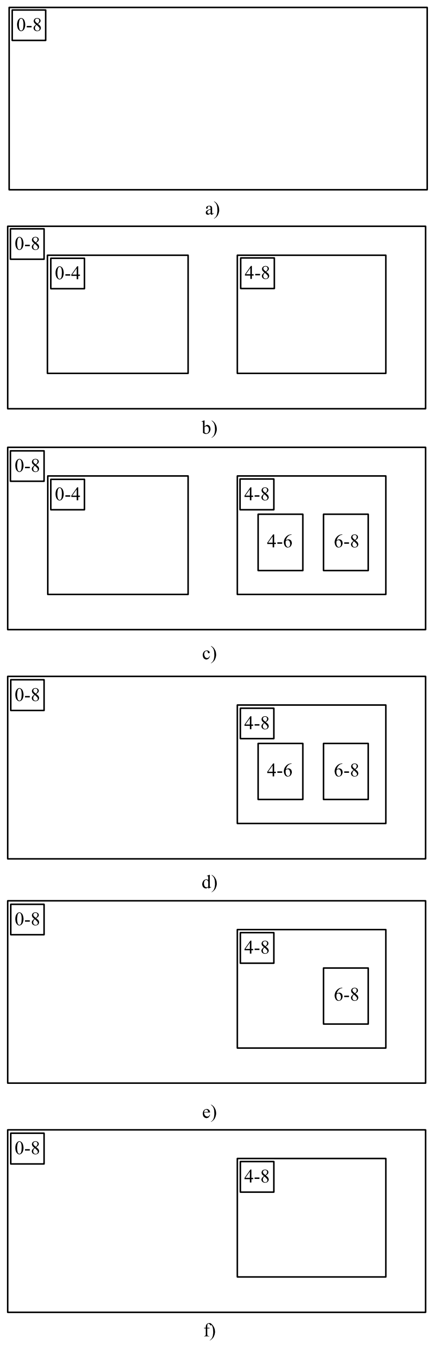
The computation of the function integral yields a large error in the interval and two nodes are created to handle the sub-intervals [0, 4] and [4, 8] as represented in Fig. 2b. The first interval needs no further refinement. However, the interval [4, 8] needs to be divided into new intervals, Fig. 2c. When the computation of interval [0, 4] ends, the corresponding node is deleted, Fig. 2d. When node [4–6] finishes its tasks, the structure becomes represented by Fig. 2e. After node [6, 8] has been removed (Fig. 2f), node [4–8] is also removed, returning the structure to the original one (Fig. 2a). This sequence of structures is just illustrative since other structures may have been created depending on the time computations will actually take.

4 D&C algorithm modeling and simulation

We detail now the description of the adaptive algorithm in the DFSS formalism, giving the implementation details in the CaosTalk environment. The integral of a function $f(x)$ in the interval $[a, b]$ can be approximated by

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{4} R_2[a_i, a_{i+1}]f,$$

Figure 2 Structural changes



where $R_2[a_i, a_{i+1}]f$ is the Simpson five-point rule given by

$$R_2[a_i, a_{i+1}]f = h_i/12\big\{f(a_i) + 4f(a_i + h_i/4) + 2f(a_i + h_i/2)$$

```
start
    super start.
    transmitionStream := Exponential new: 3 mean: 7.0.
    computationStream := Exponential new: 5 mean: 2.0.
    creationStream := Exponential new: 7 mean: 10.0.

    phase := #create.
    beta := creationStream next.

    (n > 20) ifTrue: [self error: 'Not converging'].

    self addOutputIFD: #Executive.
    self outputFunctionD: [:sol :_| sol].

    (n == 0) ifTrue: [|a m b|
        a := pa x.
        b := pb x.
        m := (a + b) * 0.5.
        pa := Point x: a y: (f value: a).
        pm := Point x: m y: (f value: m).
         pb := Point x: b y: (f value: b).
    ].
```

Listing 2 Component initialization

$$+4f(a_i + 3h_i/4) + f(a_i + h_i)\big\}$$

with $h_i = a_{i+1} - a_i$.

The Simpson three-point rule is given by

$$R_1[a_i, a_{i+1}]f = h_i/6\big\{f(a_i) + 4f(a_i + h_i/2) + f(a_i + h_i)\big\}.$$

The approximation to the integral of function f on an interval $[x, y]$ is accepted when [10]

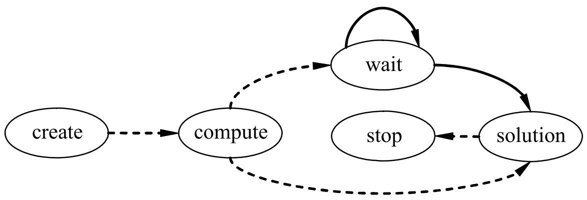
$$\big|R_2[x, y]f - R_1[x, y]f\big| < 15\varepsilon/2^r,$$

where r is the level of interval (x, y) considering that the overall interval (a, b) is at level 0 and ε is the tolerance.

Each processing node is modeled by a dynamic structure network model. The executive component is responsible for both structural changes and calculations. The initialization of this component is made by the **start** method described in Listing 2.

Three exponential streams are created to represent node creation, computation and data transmission times, named *creationStream*, *computationStream* and *transmissionStream*, respectively. The starting node ($n = 0$) computes the integral on the overall interval. The variables *pa*, *pm* and *pb* have the format of pairs $(x, f(x))$ and

Figure 3 Node phase diagram



keep the information of the minimum, middle and maximum abscissas of the integration interval with the corresponding function value. DFSS network executives as implemented in CaosTalk have access to a set of methods developed to support structural changes. These methods permit to add and remove components, to change the set of influencers and to define component input functions. Other methods are used to change network set of influencers and network output function. The methods are:

add: *aModel* **name:** *aName* **initialState:** *aBlock*, adds a component with model *aModel* using *aBlock* to define the initial state;

remove: *aName*, removes a component;

component: *aName* **addIFD:** *bName*, adds an influencer;

component: *aName* **inputFunctionD:** *aBlock*; sets the input function;

component: *aName* **removeIFD:** *bName*, removes an influencer;

addOutputIFD: *aName*, adds an influencer to the network;

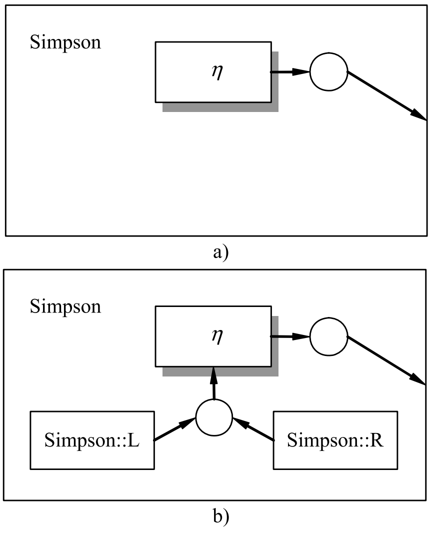
outputFunctionD: *aFunction*, sets the network output function.

Each node goes through a sequence of actions described in Fig. 3. The node starts in the **create** phase. After creation time, the node changes to the **compute** phase. In this phase, it calculates a first approximation to the integral. If the approximation is accurate enough, the node changes into the **solution** phase that models the time to transmit the solution to the parent. If the first solution has not the desired accuracy, the node creates two children and changes to phase **wait**. In this phase, the node waits until it receives both solutions and it then goes to phase **solution**. After sending the solution, the node changes to phase **stop** and it waits to be removed.

The detailed description of node behavior is given in Listing 3. To simplify the model we consider that a parent only removes its children after receiving results from both.

The method **simpson: function pa: a_fa pm: m_fm pb: b_fb** receives the function to be integrated and the three points corresponding to the extreme points and the middle point of the integration interval. The method produces a four-value array with (*err*, *int*, $p2$, $p4$) where *err* is the error estimate, *int* is the function integral, $p2$ and $p4$ are the new points computed by the method in addition to the three points received as arguments, so the Simpson five-point rule can be used. These points are sent as output values so they can be used by the children nodes, avoiding their re-computation. The Simpson component starts with a network containing only the executive, as depicted

Figure 4 The Simpson network



in Fig. 4a. The executive is created with an interval and a function. If the executive cannot calculate the integral with the required accuracy, it creates two nodes for handling the sub-intervals, Fig. 4b. The new children receive a first approximation to the integral computed at the parent node. The children nodes repeat this process recursively.

When a value arrives from one of the children, the parent node stores the result. After receiving both results, the parent node deletes the two children and sends the sum to its own parent. The ability to represent this recursive algorithm with a recursive model yields a direct representation, permitting models to be easily developed.

4.1 Simulation results

We have tested the adaptive algorithm to compute the integral of the function $f(x) = x\ln(x)(x + 1.0)^{-3/2}$, in the interval [1.0, 9.0]. The nodes used to perform the calculation are depicted in Fig. 5. The computation required 13 nodes, whereas a constant step algorithm with a step equal to the minimum step used by the adaptive Simpson's rule would require 16 nodes. We have considered the tolerance $\varepsilon = 10^{-5}$.

Table 1 gives, for several integration intervals, the number of nodes N^{A} used by the adaptive algorithm, the number of nodes N^{F} required by an equivalent non-adaptive algorithm, the value of the integral and the mean time to execute the algorithm. This time is taken as the mean of 20 simulation runs. In these experiments we have used a transmission time given by the stochastic distribution Exp(7), a computation time given by Exp(2) and a creation time given by Exp(10). These values depend actually on the computer architecture and on the load in each node. A reconfigurable computer architecture providing shorter transmission paths to neighbor nodes can increase the algorithm performance by reducing the mean time required to exchange information.

As shown in Table 1, adaptive algorithms can exhibit large gains when compared with the non-adaptive solution. In the last experiment, we can observe that although the solution tree has a height of 21 levels, only 495 nodes were used, whereas the non-adaptive version would require 2^{20} intervals. Due to the reduced number of nodes

```
delta: _ xd: xd
    | left right m fm |
    phase = #create ifTrue: [
        phase := #compute.
        ^beta := computationStream next
    ].
    phase = #solution ifTrue: [
        phase := #stop.
        ^self beta := HFSS::Infinity
    ].
    left := (name asString, '::L') asSymbol.
    right := (name asString, '::R') asSymbol.
    phase = #compute ifTrue: [ | error |
        simpson := self simpson: f pa: pa pm: pm pb: pb.
        error := simpson x.
        error < (15.0 * 1.0e-5 / (2.0 ** n)) ifTrue: [
            phase := #solution.
            out := simpson y.
            ^beta := 0.0
        ].
        phase := #wait.

        self add: Simpson name: left initialState: [:g |
            g function: f pa: pa pm: (simpson at: 3) pb: pm depth: (n + 1)
        ].
        self add: Simpson name: right initialState: [:g |
            g function: f pa: pm pm: (simpson at: 4) pb: pb depth: (n + 1)
        ].
        self component: #Executive addIFD: left.
        self component: #Executive addIFD: right.
        self component: #Executive inputFunctionD: [:ls :rs :_| Array with: rs with: ls].
        phase := #stop.
        ^beta := HFSS::Infinity
    ].
    phase = #wait ifTrue: [ | ls rs |
        ls := xd at: 1.
        rs := xd at: 2.
        ls isNil ifFalse: [
            nSolutions := nSolutions + 1.
            result := result + ls.
        ].
```

Listing 3 Simpson executive transition function

```
            rs isNil ifFalse: [
                  nSolutions := nSolutions + 1.
                  result := result + rs.
            ].
            (nSolutions < 2) ifTrue: [^beta := HFSS::Infinity].
            self component: #Executive removeIFD: left.
            self component: #Executive removeIFD: right.
            self remove: left.
            self remove: right.
            phase := #solution.
            out := result.
            ^beta := transmissionStream next
      ].
```

Listing 3 (*Continued*)

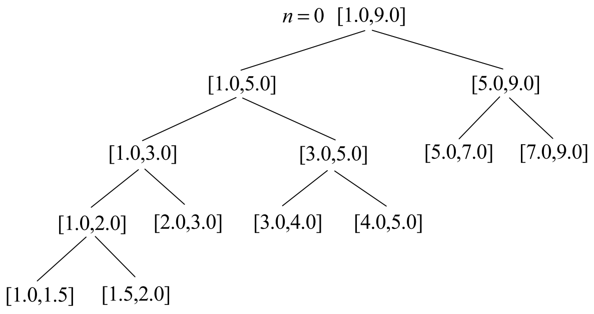


Figure 5 Nodes used for computing the integral

Table 1 Simulation results

Interval $[a, b]$	Levels	N^A	N^F	$\int f(x)\,dx$	Mean execution time (20 runs)
[1–100]	10	55	$2^9 = 512$	52.684274	196.1
[1–1000]	14	137	$2^{13} = 8,192$	309.844332	300.9
[1–10000]	18	319	$2^{17} = 131,072$	1441.010241	382.1
[1–20000]	20	401	$2^{19} = 524,288$	2234.302233	425.5
[1–40000]	21	495	2^{20}	3437.448710	448.6

used and to the parallel running of the numerical method, algorithm execution time has only a small increment with the interval width, as shown in Fig. 6.

5 Conclusions

The DFSS formalism provides a comprehensive framework for representing reconfigurable models. Modeling and simulation of parallel adaptive algorithms can benefit from modeling formalisms able to represent the dynamic creation and destruction of processes with variable topology models. The component-based characteristics of

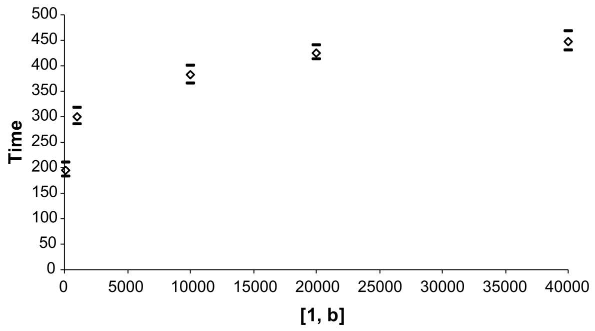


Figure 6 Algorithm execution time obtained by simulation (95% confidence intervals)

DFSS models make the formalism suitable to represent complex systems. The recursive nature of dynamic DFSS networks permits to represent the reconfigurable data structures required by adaptive numerical algorithms. We have described an adaptive integration algorithm based on the Simpson's rule that exhibits a superior performance when compared with the fixed step version of the algorithm that distributes the computational effort equally over the integration interval. The parallel adaptive algorithm was represented by hierarchical and modular simulation models with dynamic structure. This representation mimics the structural changes undergone by the algorithm and provides easy to build and to maintain models offering an excellent framework for simulating parallel adaptive algorithms.

Acknowledgements This work was partially supported by the Portuguese Foundation for Science and Technology (FCT), under project POSI/SRI/ 41601/2001.

References

1. Barros FJ (1997) Modeling formalisms for dynamic structure systems. ACM Trans Modeling Comput Simul 7(4):501–515
2. Barros FJ (2002) Modeling and simulation of dynamic structure heterogeneous flow systems. SIMULATION: Trans Soc Model Simul Int 78(1):18–27
3. Compton K, Hauck S (2002) Reconfigurable computing: a survey of systems and software. ACM Comput Surv 34(2):171–210
4. Johnston WE (2002) Computational and data grids in large-scale science and engineering. Futur Gener Comput Syst 18:1085–1100
5. Kartachev SP, Kartachev SI (1987) Analysis and synthesis of dynamic multicomputer networks that reconfigures into rings, trees and stars. IEEE Trans Comput 36(7):823–844
6. Kumaran S, Quinn MJ (1995) Divide-and-conquer programming on MIMD computers. In: Proceedings of the 9th international parallel processing symposium, pp 734–741
7. Lim A (2001) Distributed services for information dissemination in self-organizing sensor networks. J Franklin Inst 338:707–727
8. Mei B, Lambrechts A, Mignolet JY, Verkest D, Lauwereins R (2005) Arquitecture exploration for a reconfigurable architecture template. IEEE Des Test Comput 2:90–101
9. Ravasz E, Somera AL, Mongru DA, Oltvai ZN, Barabási A-L (2002) Hierarchical organization of modularity in metabolic networks. Science 297:1551–1555
10. Ralston A, Rabinowitz P (1978) A first course in numerical analysis. Dover, New York
11. Srinivas S (1992) Design and analysis of a generalized architecture for reconfigurable m-ary tree structures. IEEE Trans Comput 41(11):1456–1478
12. Wang IY (1986) Simulation of a modular hierarchical adaptive computer architecture with communication delay. MS thesis, Department of Electrical and Computer Engineering, University of Arizona
13. Yao X (1999) Evolving artificial neural networks. Proc IEEE 87(9):1423–1447

14. Zeigler BP (1990) Object-oriented simulation with hierarchical, modular models: intelligent agents and endomorphic systems. Academic, New York
15. Zhou S (1988) A trace-driven simulation study of dynamic load balancing. IEEE Trans Softw Eng 14(8):1327–1341

Fernando J. Barros is an assistant professor at the University of Coimbra (Portugal). He received a Ph.D. degree from the University of Coimbra in 1997. His research interests include theory of modeling and simulation, dynamic structure models and adaptive software architectures. Fernando Barros is currently associated editor of Simulation: Transactions of the SCS and member of the editorial board of the International Journal of Simulation & Process Modeling.