Evaluating the performance of architectures in MASCOT

Pere P. Sancho*, Carlos Juiz, Ramon Puigjaner

Universitat de les Illes Balears, Departament de Ciències Matemàtiques i Informàtica, Carretera de Valldemossa, km 7.5, 07071 Palma, Spain

Received 16 March 2004; received in revised form 30 July 2004; accepted 19 October 2004
Available online 7 March 2005

Abstract

Several design methods for software/system architectures have been developed since the origins of Software Engineering. MASCOT remains as one of such preferred design methods in the European defence arena. There are even tools that support MASCOT diagram and textual software/system designs and also their automatic code generation. However, less attention has been paid to non-functional aspects of these designs, e.g. performance evaluation. MASCOTime is a tool prototype for MASCOT, that uses the discrete-event simulation to help analysts to select among several software/system architectures. In order to provide the performance constraints of system designs, MASCOT has been annotated in a transparent way for software engineers. MASCOTime will assist software/system developers to decide which architecture matches the performance and functional requirements on early steps of design.
© 2004 Elsevier B.V. All rights reserved.

Keywords: MASCOT; Architecture; Evaluation; Performance; Simulation

1. Introduction

1.1. MASCOT

The origins of MASCOT (Modular Approach to Software Construction Operation and Test) go back to the early seventies, and in particular, to the work carried out at that time...
by Hugo Simpson and Ken Jackson at the Royal Radar Establishment (RRE) belonging to the U.K. Ministry of Defence. The spur for the creation of MASCOT came from the problems they had experienced in the software development for a large multi-computer real-time system. In other words, a large amount of code to be written for many computers operating in parallel and interacting in real time and with a large number of people engaged simultaneously on the development task that usually led to technical and management problems.

The MASCOT approach to software development, as stated in [3], contains the following features:

- “It defines a formal method of expressing the software structure of a real-time system, which is independent of both computer configuration, and programming language”.
- “It imposes a disciplined approach to design, which yields a highly modular structure, ensuring a close correspondence between design functional elements and constructional elements for system integration”.
- “It supports a program-acceptance strategy based on the verification of single modules, as larger collections of functionally related modules”.
- “It provides a small easily implemented executive for the dynamic control of program execution at run time”.
- “It can be applied to all software life-cycle stages from design onwards”.
- “It can form the basis for a standard system of software procurement and management”.

MASCOT is a design and implementation method for real-time software development and it brings together a coordinated set of techniques for dealing with the design, construction (or system building), operation (or run-time execution) and testing software.

At the heart of the method [15] there is a particular form of software structure supported by complementary diagrammatic and textual notations. These notations give visibility to the architecture design as it emerges during development, and implementation takes place via special construction tools that operate directly on the design data. Design visibility greatly eases the task of development management, and the constructional approach ensures conformity of implementation to design, and allows traceability of real-time performance back to design.

Fig. 1 shows the basic blocks for constructing MASCOT designs and diagrams. Activities model processes or threads of execution. All activities in a MASCOT design run in parallel but their execution code is strictly sequential. For two activities to communicate an intermediate route is needed. There are three kinds of routes, namely: Channels, pools and signals. Each route implements some kind of communication protocol:

- **channel**: Producer–Consumer protocol. A producer activity puts a data item into the channel and another consumer activity gets the data from the channel.
- **pool**: Readers–Writers protocol. Here, when a writer produces a data item and stores it in the pool, it overwrites the existing data item. Once stored the data item is available (as some sort of reference data) to all readers interested. Readers make...
a copy of the information rather than completely eliminating it from the route as in the case of the channel.

**signal**: Combines features of the two routes already described. Data items written on a signal overwrite the (possible) existing ones, so that only the newest data is present. But data items can only be read once for all readings remove the item from the route.

Along with the communication issues these protocols carry some dramatic performance implications. The study of the performance characteristics of each MASCOT component is part of the background research that supports this article.

MADGE is a construction tool that produces MASCOT designs and automatically generates Ada code. Thus, MADGE and other tools may be used to consider different software/hardware architectures on design but unfortunately they cannot be evaluated. MASCOTTime [13] is a prototype of discrete-event simulator for MASCOT designs that shows the difference in performance among architecture candidates in early stages of software/system construction.

In no way can MASCOTTime be regarded as a handle-cranking technique, guaranteed to produce, painlessly, the solutions of complex problems. Success is achieved by using a modular decomposition of MASCOT approach which is relevant to management, design, construction and execution, and which allows problems that contain complexity to be partitioned into smaller pieces. Design is essentially a creative process requiring the application of skill and experience. MADGE provides the process with visibility and code generation, which gives control over it, and MASCOTTime provides the performance evaluation of such designs.

The future of no method can be predicted with certainty. Nevertheless, one interesting feature to be added would be the performance evaluation of the design elements in early stages of the system design by incorporating the time requirements and constraints. This article is concerned with an extension to the MASCOT method that makes performance evaluation of the designs easier.
2. MASCOTime

The software/system architecture modelled with MASCOT is based on data flow concepts. The simplicity of its design methodology has several advantages, e.g. allowing the distribution of system functionality to be represented. This textual/diagrammatic representation provides the means for controlling functional interactions and their propagation among the components. However, there is no control in one important non-functional feature of the system that is being modelled—the performance of the software components and even the performance of the overall system. MASCOTime is a discrete-event simulator for MASCOT designs that mainly describes the service time of the functional components and analyzes them. Therefore, performance annotations for the components have been added to MASCOT in a friendly manner, which allows derivation of a simulation model of the system’s architecture. These annotations incorporate the timing and capacity constraints that the future system would require. The consequence of taking into account the performability of the system as well as the functionality at its stage of design gives the opportunity to choose among different architectures once the simulation predictions show the advantages or drawbacks of each one. In this way, software/system analysts may choose which architecture fits better to hold both kinds of requirement. In order to minimize the training on MASCOTime issues, the extension has been developed avoiding an intermediate interfacing between the design with performance annotations and the simulator. Thus, MASCOTime components and simulator objects are isomorphic.

2.1. Discrete-event simulation

Discrete-event simulation concerns the evolution over time of a system. The system is represented—modeled—by the so-called state variables [8]. The state variables contain all the relevant information of the system, and discrete-event simulation consists of manipulating their values at certain points of time. All the changes to the state variables must be done according to the logic of the simulated real-world system and only when an event occurs. An event is defined as an instantaneous occurrence that may change the state of the system and the time of this occurrence is calculated with the help of a pseudo-random variable generator. In this way, the variables of the system do not change continuously in time but at discrete points in time.

2.1.1. Structure of a simulator

Every simulator has to have the following data structures:

- **Simulation clock.** This is a variable giving the current simulation time.
- **Event list.** A list containing all the forthcoming events, possibly ordered in some way that helps localizing the foremost.
- **State variables.** A set of values which model the simulated system, together with routines for manipulating them and calculating further events from them.
- **Statistical counters.** Variables containing occurrences of certain magnitudes of interest.

The main program has a very simple way of working. First it reads the first event from the list and actualizes the simulation clock to the occurrence time of the event. Then, it changes
the state variables according to the type of event. These changes on the state variables
may generate some other events which are added to the Event list. Finally it updates the
statistical counters. This process repeats until the end of the simulation.

3. MASCOT extensions

Whereas any MASCOT architecture observes time effects for process interaction, it
does not provide a way to ensure good performance or even to have a rough prediction of
the response time of the software components or even the whole system.

The objective of this paper is to present a possible extension to the method which
involves performance extensions, simulation of the system, performance results, and a
whole new way of working with MASCOT architectures.

3.1. MDL and MASCOTime

MASCOTime is embedded into the MDL files which describe the system design.
MASCOTime code is inside \{\* \*\} comments so that it does not interfere with the normal
operation of the MDL files and completes the definition of the elements declared in each
file of the MDL description of the system with performance annotations.

Fig. 2 shows the appearance of a MDL description file. The MASCOTime code is
included at the bottom of it, delimited by \{\* \*\}. Each component is described twice, the
first time in MDL and the second in MASCOTime. This structure is somewhat redundant;
it may seem a better idea to write the MASCOTime code next to the component it belongs
to, but our goal was to interfere as little as possible with the existing software engineering
work-flow. In this way, although there are two definitions of the same component, each one
describing a different aspect, the MASCOTime part can be easily ignored by the designers
whenever they want to concentrate on the functional architecture.

Let us have a look at what Fig. 2 describes. As stated in line 1 the code describes a
subsystem. By line 3 we know this subsystem provides a GET interface (defined somewhere
else) to the outside. That is the only way to communicate with the subsystem. On line 5
some imports are requested for the definitions to come. On lines 7 and 8 an activity is
instantiated with the name A_INTERRUPT_CLOCK. An activity is one of the simple pieces
that MASCOT uses to build programs. It represents some kind of process or task. The
complete definition of the activity template, CLOCK_INTERRUPT, resides in another file,
and we have imported it on line 5. On line 8 a connection is being made between P1 from
the activity and W1 from CLOCK_INFORMATION (which has not been declared already).

On line 9 is the definition of a ROUTE, i.e. a common space for two activities to
communicate. In this case it is a SIGNAL named CLOCK_INFORMATION (and now we know
who line 8 was referring to). A signal is another primitive element of MASCOT, which is
characterized by having the space to keep one data item, destructive writing on it and
destructive reading. That means all attempts to write on the signal will overwrite any
existing data item, and all reading operations will delete the data item from the signal.
So a writer cannot be blocked but a reader will if there is no data item to read. This kind of
interaction between processes is what the MASCOT architecture provides.
Fig. 2. MDL code with embedded MASCOTime extensions.

Line 11 states that the outside connection defined in line 3 is connected to \( W2 \) of the signal. And that is all that is needed to describe this simple subsystem. Of course, definition files can grow to any complexity but the syntax will not vary.

Lines 13–29 are the added MASCOTime definitions. They are what this paper is concerned about, because they are the bridge to performance evaluation. The first block of definitions, lines 16–19, describe the signal again. But this time other aspects are observed—performance aspects. Line 17 gives the signal a constant reading time of one unit of time. Line 18 makes the writing time of the signal constant with value 1.0 unit. This is all that is needed to perform a simulation with the signal: we need to know how long it takes to read from it and to write on it. Other functionality aspects are given by MASCOT, i.e. the destructive writing, etc.

The second block goes from line 21 to line 26 and completes the definition of the activity. An activity is defined by its sequence of actions. Lines 23 and 24 are two actions that will be repeated through all the simulation in an endless loop. First the activity waits a constant time of 10.0 units, line 23. Then it writes something on \( \text{CLOCK\_INFORMATION} \) which is the signal described. As soon as the writing is complete (exactly 1.0 unit of time after, by the definition of the signal, line 18) the activity starts over again and begins waiting 10.0 units of time, and so on.
In the example code of Fig. 2, attributes and operations get their parameters as distributions. Determining those values is the most difficult part about the extensions. Some can be looked up in the technical sheet of a component. That is the case of some hardware devices the program uses, e.g. a hard disk or a screen, where the store time or the refresh rate can be easily found. The times for software devices can be deduced from the algorithm that implements their data structures and experience definitively plays an important role in figuring out their distribution. Measuring similar existing programs is also a convenient way to have good approximation of what can be expected from the future system. For a software performance engineer this is common work, and for that reason fairly good techniques have been developed for that purpose.

In a higher level description the example of Fig. 2 consists of a clock that puts ticks inside a channel every 10.0 units of time. The MASCOT’s graphical representation of this is shown in Fig. 3.

To define more complex systems more MDL files are needed and MASCOTime code has to be specified for each element. This can be done at early stages of the design life-cycle.

3.2. MASCOTime to simulation

Once a software performance engineer has added all the MASCOTime code for all the components in the system there is one more step to do before the whole system can be simulated. A translator transforms the MASCOTime code embedded in the MDL files (MASCOT Description Language) into simulation code. The work is done automatically and a Main.java file is generated.

The Java file is linked with the MASCOTime simulation library and when compiled and run will generate simulation results on a text file as well as a log of everything that happened in the system during execution (in case it is desired). This performance data comes with confidence intervals and component-level detail.

This translator checks not only the syntax but also the semantics of the MASCOTime code written. It is capable of reporting the usual errors. These include duplicate (or missing) definitions, wrong parameters, type mismatches, etc.
3.3. Deciding

A performance engineer can tune the parameters of the simulated system, i.e. the MASCOTime code, and run the simulator again until the results are satisfactory. When that condition is eventually achieved the engineer will have a set of approximate component-level performance requirements. With these in hand it must be decided if the architecture is suitable or not.

In some cases the performance requirements of some components will be unreachable and that distinguishes an eligible architecture from an impossible one. For example, a channel (a bounded fifo queue in MASCOT) can be easily programmed to have a bigger capacity, the only cost being on the memory used; but it is rather difficult for a hard drive to retrieve data in less than a millisecond. The latter case could show an impossible architecture and must be discarded.

3.4. Java MASCOTime simulator

Although the simulator is a complete one with all desirable features, it is used much like a simulation library. As stated above, a Main.java file can be generated with a main procedure which imports the objects defined in the simulation library. This main procedure is intended for declaration of the components only, because it is, in turn, a method in one of the library objects who does the actual simulation. The customizable main procedure only defines the objects and starts the simulation.

In order to produce reliable results the simulation libraries have two modes of functioning, transient and stationary. In transient mode the simulator outputs the evolution of the mean of every measurement so that the sequence can be represented graphically. The curve represented will clearly show a transient or warm-up zone and a stationary one. Once delimited the initial transient the real simulation can start. In this mode the simulator skips the measurements belonging to the initial transient so the overall means are not biased and the confidence intervals calculated are reliable.

An inner view of the simulator shows classes which are isomorphic with the MASCOT elements, i.e. there is an Activity.class, a Signal.class, a Channel.class, and so on.

Going deeper into the implementation of the simulator would exceed the purposes of this paper, but a complete description can be found in [12].

4. Case study: Simple radar management system

Suppose a model is needed to build a software to manage two radar stations. Every data sent by the radars must be stored in a disk drive and shown on a screen. One possible MASCOT architecture for such system is shown in Fig. 4.

In Fig. 4 there are two radar stations modeled as two MASCOT activities which send their information to two dispatchers via two channels. The dispatchers read the information and send it to the disk and to the display after having formatted it for each media. The disk subsystem is modeled by an activity with an intermediate channel which represents the disk queue. The display is a MASCOT server because it outputs information from the system.
It is very easy to draw the MASCOT diagram shown in Fig. 4 with some tool specially designed for that purpose, for example MADGE. The tools can generate MDL code and this can be completed with MASCOTime definitions. The next lines show all the MASCOTime code embedded in the MDL system definition of Fig. 4.

```
SYSTEM RADAR;
USES RADAR, DISPATCHER, DISK, DISPLAY, CHANNEL;

ACTIVITY A_RADAR1 : RADAR(P1=CH_RADAR1.W1);
ACTIVITY A_RADAR2 : RADAR(P1=CH_RADAR2.W1);
ROUTE CH_RADAR1 : CHANNEL;
ROUTE CH_RADAR2 : CHANNEL;
ROUTE CH_DISK : CHANNEL;

ACTIVITY A_DISPATCHER1 : DISPATCHER
   (P1=CH_RADAR1.W2,
    P2=S_DISPLAY.W1,
    P3=CH_DISK.W1);

ACTIVITY A_DISPATCHER2 : DISPATCHER
   (P1=CH_RADAR2.W2,
    P2=S_DISPLAY.W1,
    P3=CH_DISK.W1);

ACTIVITY A_DISK : DISK (P1=CH_DISK.W2);
SERVER S_DISPLAY : DISPLAY;

/*
BEGIN MASCOTIME
   DEFINE CH_RADAR1 : CHANNEL
      CAPACITY(4);
      TREAD(CST(0.1));
      TWRITE(CST(0.1));
   END DEFINE;
   DEFINE CH_RADAR2 : CHANNEL
      CAPACITY(4);
*/
```
The code shown above is read by the MASCOTime translator which generates a `Main.java` file with simulation code for this particular system. Once again, to complete the MASCOTime extensions the duration of all the operations have to be given. In the case of radar stations the parameter makes the radars scan every 50 ms because that is the real rate they will have once built. Channels are such simple structures that reading
and writing operations will surely be fast. On the other hand interpreting and formatting
the information—a task done by the dispatchers—can be quick some times and more time
consuming some others. Being a little pessimistic we approximate them with exponential
distributions. Any user of these tools must be aware of the danger of using imprecise
figures.

The next step is launching a simulation in transient mode and observing a graphical
representation of the results. In Fig. 5 we can see the evolution curves of one the channel’s
queue length. It is the average of 100 runs of the simulation so we can be fairly sure that
when curve stabilizes the initial transient is over. Of course the division point cannot be
determined accurately but by Fig. 5 we may approximate that by 60 the system is stable.

With this last value in hand we can move onto the real simulation. It is real in the sense
that its results are accurate and without bias because only the measurements beyond 60
will account for the calculation of the means and confidence intervals. Thus, we are not
influenced by the warm-up period.

Throughout the example, every time a simulation is performed it must be assumed that
all this process has been carried out, although it is not described in the text. Making the
full description would add no value to the text.

Fig. 6 shows a selection of the simulation results, where the channels are almost empty
most of the time: the system is underused. At this point we can leave the system as is, but
the interesting use of these tools is that it is possible to change and test other architectures
to see if maybe simpler ones still work well. The first change that can be made concerning
the architecture is to join the two channels into one and the two dispatchers, too. So the
new architecture is the one shown in Fig. 7.

Eliminating one dispatcher and one channel from the previous code (and doubling the
capacity of the remaining one) and simulating again the results are no surprise. The system
can deal easily and at a good rate with the radar data. The radar channel is full only at
4.4%; see the simulation results in Fig. 8. It has been tested with the simulator that this
configuration can stand well up to 12 ms between radar scans, that is, 83 scans a second,
so a 700 mph plane will have been scanned 200 times before it flies half a mile. We have
simplified the original architecture to another that is cheaper to implement and easier to
maintain.
channel Radar Data 1
=================================
Mean queue length:
0.001996, 0.0498999% full

channel Radar Data 2
=================================
Mean queue length:
0.002005, 0.050133% full

Fig. 6. Fragment of the first simulation run.

channel Radar Data
=================================
Mean queue length:
0.179719, 4.492984% full

Fig. 8. Fragment of the second simulation run.

Which of the two is more scalable in case we want to add more radar antennas in the future? We just have to copy the antennas in the original code, say five times, and simulate again. The channel is at the 96.4% of its capacity, see Fig. 9, a very dangerous point. The simulation has been able to preview the writing conflicts among all five radar stations: writing to the main channel takes 14.71 ms (see Fig. 9) where the operation itself lasts 0.1 ms (as defined in the MASCOTime extensions). They are waiting 14.6 ms in conflict with all the others. Maybe in this case the first schema would give better results.

To have the original schema simulated more work is needed because the channels and the dispatchers have to be replicated resulting in a model full of components. This solution
Fig. 9. Fragment of the simulation with five radar stations.

Fig. 10. Second simulation with five radar stations.

again is not scalable to five radar stations. The channels are completely saturated, as it is shown on Fig. 10. Let us look for the cause and for a solution.

The simulation results help us track the source of this saturation, i.e. the bottleneck. The ease and the speed to make changes in the parameters and obtain new simulation results
<table>
<thead>
<tr>
<th>Channel</th>
<th>Mean Queue Length</th>
<th>Fullness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radar Data 1</td>
<td>0.002001, 0.050044%</td>
<td></td>
</tr>
<tr>
<td>Radar Data 2</td>
<td>0.002003, 0.050080%</td>
<td></td>
</tr>
<tr>
<td>... until Radar Data 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disk Queue</td>
<td>0.422309, 2.111546%</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 11. Simulation results with an intermediate buffer.

gives all the facilities to make statements and also to find solutions. Let us first follow the path to the bottleneck.

As things went well before we know the dispatchers could handle the traffic. Now it takes 53.78 ms to write in the channel of the disk (see Fig. 10), and it should last 0.1 ms. Again, looking at the simulation results we discover its channel completely full. So it is the disk who slows the system down. It is necessary to send fewer requests to the disk in order to desaturate its channel.

One way to loosen the requirements of the disk channel is, perhaps, setting up a faster disk. That is related to the hardware architecture, and it would reduce the storing time which now is 12.4 ms. Looking at the specifications of the most well-known disk manufacturers this time can be reduced to half, at a very high cost in money. And this action still does not solve the problem. If one encounters this problem once the system is finished the first reaction will be to change the disk, and that will not fix things up—according to a simulation not shown here. This is to show that there are some architectural problems that inherently perform badly and cannot be solved no matter how fast the hardware is. So, we will have to look for other solutions, now that we have not built the system yet.

One old tool when dealing with disks is buffering. So let us add this architectural component to the system. This can be done in two ways. The first is programming the activities to have a buffer and store five readings from the radar before sending them, as one big message, to the disk. The second way is to define what in MASCOT is called a generic IDA, which is a somewhat more complex data structure, and put it instead of the disk channel. The IDA will only send to the disk groups of five petitions if it is programmed to. Both solutions can be programmed in MASCOTime but attending to the material explained in this paper we will use the first one.

Building all these changes into the code and re-running the simulator we can see that the buffer is of great help. Of course the reading and writing operations of the disk channel have been specified to last five times longer. Despite all this now the system works perfectly, see Fig. 11.
This process of setting architecture performance tests can last as long as needed, until the designers are able to find a suitable architecture to meet the performance requirements.

In this section, the process of making architectural decisions with the help of the MASCOTime simulator has been shown. A first design has been simplified with the assurance that it will not lead to bad performance. Then we have looked for scalability and by making simulation runs we have seen that none of the models was scalable, and that a buffering system (or any equivalent solution) was needed. Thus a necessity has been foreseen that could affect the design decisions.

The MASCOTime simulator is useful to help take performance decisions, but there are still many other considerations concerning the choosing of an architecture. Here, we intend to provide a tool for the performance aspect only.

5. Conclusions and future work

This paper has shown the capabilities of a new prototype extension for MASCOT design methodology known as MASCOTime. MASCOTime provides a discrete-event simulator for MASCOT designs in order to select among different software/system architectures. MASCOTime code is added as comment annotations inside the original MDL file (MASCOT Description Language) in a friendly manner. In this way the software engineering process is not disturbed and the annotated code provides, with some redundancy, a functional description of the system together with its performance constraints. On the other hand, a translator transforms the MASCOTime description into a `Main.java` file that is linked with the MASCOTime simulation library. Once the file is compiled and run, it will generate simulation results on a text file as well as a log of every event in the system that is being designed. The approximate performance evaluation through simulation may be used by software engineers for choosing among alternative architectures in early phases of software/system development. The joint architectural description of a new system into a reduced set of functional and performance aspects eases the tasks on its ultimate construction.

The progressive refinement of the architecture with the MASCOTime tool does not interfere with the normal work-flow of the MASCOT users. A designer can do as much refinement as considered necessary but can stop at any moment and move on to the next regular step of the software development process. This is considered a good feature because it does not impose an additional task to software developers.

There are some points that are still left for future research and development. The main tradeoff while working with the MASCOTime tools is the treatment of the initial transient. Some further efforts have to be done in order to have the simulator skip automatically the initial transient state so that only one run has to be done to obtain similarly accurate simulation results.

As far as the routes are concerned, MASCOTime extensions are enough to define simulation models. However, when dealing with active elements, possibilities grow quickly. We have presented the current set of implemented actions available, but future extensions are being researched. Firstly, never-blocking versions for reading and writing actions; in this way, the active element will not be blocked when trying to read from an
empty route. Instead it will return and proceed with other actions. This implies some kind of conditional branching and leads to the implementation of control statements e.g. IF ... THEN ... ELSE, WHILE, FOR, etc. which would bring some more flexibility in the MASCOTTime definition of a sequence of actions. The extensions just introduced have not been integrated in the current system because they would force the designer to go a little too deep into implementation details. This was not desired because one of the main goals was to minimize the interferences to the work-flow of the developers who work with this methodology. The ability to perform more detailed simulation improves usability as the design advances, but for an early evaluation the current set will usually be enough.

References