Overload Handling in Soft Real-Time Systems:

A Case Study using ROOM/ObjecTime

Alexandre Nikolaev

A Thesis

in

The Department

Of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

April 2003

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-77718-9

Canada

ABSTRACT

Overload Handling in Soft Real-Time Systems:

A Case Study using ROOM/ObjecTime

Alexandre Nikolaev

In real-time systems, deadlines are imposed on the response time. Not meeting a deadline in a hard real-time system is equivalent to its failure, while in soft real-time systems occasional minor delays in responding to events are acceptable. Only when the delays are frequent or considerable, performance degradation up to system malfunction can be observed. The developers for real-time systems must therefore pay special attention to the performance of the system. The scheduling of tasks in the system becomes critically important as it directly affects the system performance.

While, traditionally, real-time system developers have used low-level software programming paradigms, the rising complexity of real-time software is creating a demand for CASE tools that allow for development using a combination of visual modeling and design, augmented with code-segments. One such tool is the ObjecTime Developer based on ROOM (Real-Time Object Oriented Modeling) development methodology.

In this thesis we study usability and effectiveness of this tool for building a soft real time system with special attention on the performance and behaviour of the system under overload conditions. As a working example we develop a radar simulator system, which observes air targets whose speeds, number and distances are constantly changing; thus, creating a constantly varying load of the entire system, with dynamically appearing objects. The major contribution of this thesis is developing custom overload handling policies to improve performance and illustrating how they may be implemented within the framework of the tool. The native scheduling policy of ObjecTime, the classical priority-based policy and our own policy based on the period-of-execution are studied.

ACKNOWLEDGMENTS

I would like to express many thanks to Dr. Manas Saksena, who was guiding me not only through the entire work on this thesis, but also towards new interesting ideas. I also appreciate his patience and ability to explain things in a clear and simple way. I also thank him and his wife Roli for the warm welcome in Pittsburgh, when I was there to finalize the work.

I wish to thank Dr. Jayakumar for his willingness to be my co-supervisor and to help me in many other ways. His guidance for the last steps of defence preparation and thesis submission was of great value for me.

I would like to thank all the members of my examination committee for providing me with valuable feedback and for pointing out important errors and omissions in the text.

I am also thankful to my wife, Marzena Oczkowska, who read my work many times over and over, pointing out stylistical errors and difficult to understand places, even after the thesis was officially proofread. I also appreciate a lot her help in terms of doing most of the housework, while I had rush hours working on the thesis.

Finally, I wish to thank my parents, for constant support throughout my life, teaching me how to think and giving me a chance to get an education. Additionally, I thank my father in particular who, as an electronic warfare engineer, gave a number of explanations and important key points concerning real-life design and operation of radar systems.

TABLE OF CONTENTS

List of Figures

List of Tables

1 Introduction

A real-time system is a system where the correctness of computation depends not only on the result produced, but also on the time at which the result is produced. Thus, real-time systems impose deadlines on the response time – the time taken to process input signals/events and produce corresponding output signals/responses.

Real-time systems (or, more accurately, timing constraints in real-time systems) are often characterized as hard real-time and soft real-time. In hard real-time systems any delay beyond the specified deadline is considered a system failure and could lead to disastrous consequences. By contrast, soft real-time systems do not have such strict timing requirements. When timing requirements are specified as deadlines, the occasional missing of a deadline will typically result in system performance degradation – not system failure. In other words, the overall system performance is often a function of the delays in responding to events – regular delays beyond the expected deadlines can lead to significant performance degradation, and even making the system unusable for its intended purpose.

The development of software for real-time systems has always posed special challenges to the developers since they not only have to worry about functional correctness, but also have to treat real-time performance issues as a primary concern and make sure that the system behaves as desired under a range of operating environments.

Over the years, academics and researchers have done extensive research in real-time systems, looking at techniques to help designers meet the timing requirements. A significant body of research has gone in scheduling techniques and related analytical methods to analyze and predict the ability of a system to meet its deadlines. Most of this research has focused on hard real-time systems, partly because the problems are more tractable, and partly because the problems are more well-defined. As a result, while systematic methods exist to

deal with timing requirements in hard real-time systems, developers often resort to more ad-hoc techniques for soft real-time systems.

1.1 The Rising Complexity of Real-Time Software

The importance of timing requirements in real-time systems has led to a traditionally conservative mentality in many real-time software developers. This manifests in the developers' desire of having total control over how the hardware resources are allocated to application tasks. As a result, real-time software developers often use only low-level mechanisms (e.g., a small operating system kernel with a minimal assembly/C execution environment) that give them full control of how the hardware resources are used by application programs.

As hardware has become cheaper and faster, it is now possible to migrate an increasing amount of functionality from hardware to software. With the many advantages of cost and flexibility afforded by software implementations, this trend has continued heavily in real-time embedded systems. As a result, the traditional minimalist real-time programming model used by many developers is difficult to sustain and they have to resort to higher-level software construction techniques and tools.

One such trend in sophisticated software development tools for embedded real-time software is the use of tools that provide a modeling and design environment using object-oriented modeling and design methods. Examples of such tools include Rational Rose and Rational Rose Real-Time from Rational Software (and its precursor ObjecTime Developer from ObjecTime Ltd.), Rhapsody from iLogix, Tau from Telelogic, and Artisan Developer Studio from Artisan that are all based on Unified Modeling Language (UML) as the modeling and design language.

A distinguishing feature of such tools is that they allow the developers to build significant software functionality using high level design models, and augmenting these design models with code-fragments where needed. In conjunction with automatic code-generation from such high-level design models,

the developers are saved from the trouble of developing "infrastructure code" such as code that interfaces with the operating system, implements state machines, implements communication structures, etc.

In this thesis, we study the effectiveness of such design methods and tools for the development of real-time software. The tool chosen to experiment with was ObjecTime Developer — a leading object-oriented modeling, design, and code-generation tool from ObjecTime Ltd.[1]

1.2 Radar Target Tracking System Case Study

A significant objective of this thesis was to study how such design and code-generation tools can be effectively employed in the design and development of significant real-time systems. We wanted to see not only how the designer could effectively implement the functionality of a real-time system, but also whether the tool provided enough flexibility for the developer to exercise control over how resources get allocated between application tasks.

The system we decided to use to perform this study was a radar target-tracking system. The target tracking system of a Radar system is responsible for tracking flying targets within the radar's monitored space. The radar antenna provides inputs to the target tracking system giving location of objects in its field. The target tracking system makes use of analytical predictions to identify the targets and how they are moving within the field. The output of target tracking system can be other systems (e.g., a weapons system) or human controllers.

The most important concern in radar target tracking systems is to accurately track the target objects in the field. Unfortunately, the load on the system grows as the number of targets increase. When the system is unable to

[1] Much of the work for this thesis was carried out in 1998-1999 time-frame. At that time, ObjecTime Developer was the leading design tool in this category. The company ObjecTime was later bought by Rational (which has now been bought by IBM) and the tool morphed into Rational Rose Real-Time, which is a leading object-oriented design tool for embedded/Real-time systems. Many of the ideas of the modeling language ROOM, used by ObjecTime, are also merged with Unififed Modeling Language (UML) or its extensions.

successfully track all targets, it must nonetheless make an attempt to track the most important targets.

The radar tracking system provides for a good system for experimentation as it is a non-trivial system and is a good example of a system with soft real-time timing constraints. Additionally, it provides many challenges to the real-time software designer in balancing the overall system performance goals with the available computing power.

1.3 Goals of the Thesis

The high-level goal of this thesis is to implement a significant case-study of a soft real-time system (radar target tracking system) using a CASE tool (ObjecTime Developer) with the aim of evaluating the effectiveness of such tools in the design and implementation of soft real-time systems. We were interested in evaluating the effectiveness of such a tool from two perspectives, as outlined below.

First, the purported aim of these tools is that they allow the user to focus on abstractions and problems that are closer to the application domain saving them from the tedium of low-level implementation concerns. To do this, the case-study chosen for implementation was sufficiently complex that it would have required significant effort if it was to be implemented using standard C/C++ abstractions. A significant part of this evaluation was to see if the tool provided enough flexibility to the user from a real-time perspective. One of the requirements imposed on the radar target tracking system was to make the implemented system behave well under overloaded situations -- a common concern in a soft real-time system. The evaluation methodology for this purpose was ad-hoc – primarily to see if the tool and/or the development paradigm supported by the tool created any significant hurdles in the development.

The second, and perhaps more interesting goal was to develop explicit mechanisms and policies to control usage of resources under overload situations and to see if we could develop and implement an overload control policy that

would outperform the default behaviour provided by the tool. To achieve this goal, the case-study had explicit requirements on managing behaviour under overload and to assess the performance of the system under varying load conditions. We utilize and compare performance of three different scheduling techniques: ObjecTime native scheduler, classical priority-based explicit scheduler and our own period-of-execution-based explicit scheduler. The last one, as the name suggests, assigns different tracking periods to targets of different 'importance'.

1.4 Related Work

A few research papers have reported case studies of using ROOM methodology involving extensive use of ObjecTime developer. Several of these papers have tried to see the suitability of ROOM and ObjecTime Developer as a methodology for development of hard real-time systems.

In [14] Saksena et al. showed how ROOM methodology could be applied for building embedded real-time systems, using an example of a car cruise control. Showing high suitability of ROOM in general and ObjecTime in particular for such designs, they recognize, however, the difficulties associated with using priority mechanisms. Those include priority assignment decisions and priority inversion problems. This research was further extended, while also using a case-study of automobile cruise control by Rodziewicz in his thesis work [12].

This work was later extended in [13] by modifying the run-time system of ObjecTime to address the concerns related to priority inversion problems identified in [14]. Using a fairly detailed case study on a Train Tilting System, they showed that with the proposed modifications ROOM and ObjecTime could be used effectively for developing hard real-time systems with predictable behaviour in terms of possibility to perform timing and schedulability analysis.

Object-oriented design methodology using ObjecTime developer was also studied by G.Krasovec et al. [8] who studied the target tracking mechanism of a sonar. Advantages of ObjecTime and ROOM modeling were shown, such as:

component distribution and concurrency, integrated behaviour modeling, representation of external components and, also very important designer aid – substantive code generation.

Many researchers have studied the problem of ensuring controlled and graceful degradation of system performance under overload conditions. This is especially relevant in the construction of soft real-time systems. Several researchers have used an approach to dealing with overload based on value or importance functions. A good overview of this approach is presented in Burns et al [6]. They used the case study of an autonomous vehicle control system and their value function was based on pair-wise comparisons.

Various value-density algorithms, such as Best Effort, Simplified Rolling Horizon scheduling and Dynamic Priority scheduling were presented by D.Mosse et al [11]. Dynamic Value Density was studied by S. Aldarmi et al in [1]. They showed the advantage of Dynamic Value Density over Static Value Density and Earliest Deadline first scheduling schemes in overload situations.

There have been several studies that have addressed the problem of target tracking in radar systems as well that are very similar to the problem chosen in this thesis. Clark et al. were building an airborne target tracking system prototype for USAF AWACS program [7]. They used a notion of target importance and divided all the targets dynamically into three importance groups. They used Quality-of-Service to monitor and subsequently control application operation. The system performed extremely well during AWACS demonstrations.

Similar to [7], work was done by Lee et al in [10]. They introduced a concept of "service classes" for a radar system. They distinguish between "more critical" and "less critical" targets by placing them dynamically in service classes, each of those utilizing different amount of computation resources. In their approach, they also limited the number of service classes to three, thus a target can get only one of three fixed values of computation resources, according to its "importance".

1.5 Thesis Contributions

This thesis presents further studies of ROOM and ObjecTime usability and effectiveness, but focusing primarily on the design of soft real-time systems, and especially the ability to control performance behaviour under overload. A simulated radar target-tracking system is implemented and used as a case-study to experiment with overload handling.

To deal with overload handling, we extend the notion of task importance, developing a somewhat continuous gradation scale for tasks, as opposed to discrete importance classes presented in previous works. We vary this importance dynamically attaching also generic overload property to the scheduling process. Thus, it gives us ability to have smoother distribution of computation resources between targets and "fine tune" them according to overall system load.

Two different overload handling policies are proposed and implemented within the framework of the tool. A generic overload handling policy is implemented by dynamically varying priorities based on currently estimated importance. A more "intelligent" overload handling is proposed based on modifying the rate at which different targets are tracked – depending on the (current) importance of the target.

The two proposed policies are evaluated through experimentation and objective assessment of system performance. The performance is compared to the baseline where no special mechanism is implemented to deal with overload or task importance. We show that using smart scheduling techniques we can indeed optimize the performance of radars.

The results and observations of this work can be applied in different research areas.

1. It clarifies the usefulness and effectiveness of ROOM methodology and ObjecTime (and/or similar tools) by showing how and to what extent they can be applied in the design of real-time software.

2. At the same time, it points out some drawbacks and lack of features which can be helpful to the developers of such CASE tools.
3. It gives an aid to real-time system designers in general by showing how custom scheduling policies and mechanisms can be successfully applied to soft real-time systems to improve system performance when compared to generic mechanisms.
4. It continues a research of value property of objects in real-time systems, by extending the idea of importance groups to somewhat contiguous importance functions.
5. It could be helpful for radar designers in particular, by showing how ROOM methodology with appropriate CASE tools can be applied to the creation of such systems, and by presenting them value functions for this particular application to consider.

2 Background

In this chapter, we present the relevant background information that this thesis relies on. In Section 2.1, we discuss briefly the ROOM development methodology. Then, in Section 2.2, we take a look at the existing real-life radar systems.

2.1 ROOM Development Methodology

ROOM has originated from the telecommunications community, and has been successfully applied to many commercial systems through the supporting CASE tool ObjecTime. ROOM provides features such as object-orientation, state machine description of behaviours, formal semantics for excitability of models, and possibility of code generation.

The ROOM method adopts an operational approach to system analysis, design and implementation. It is based on establishing early operational models of the system and then refining them to implementation. It uses the concept of executable models which evolve from requirements to design to implementation. A ROOM executable model is a set of coherent structure and behaviour views which can be compiled and executed on a variety of simulation and/or target platforms.

Modeling of systems with ROOM is performed by designing *actor*s, which are encapsulated concurrent objects, communicating via point-to-point links. Inter-actor communication is performed exclusively by sending and receiving *messages* via interface objects called *port*s. A message is a tuple consisting of a signal name, a message body (i.e., data associated with the message), and an associated message priority.

The behaviour of an actor is represented by an extended state machine called a *ROOMchart*, based on the statechart formalism. Each actor remains dormant until an *event* occurs, i.e., when a *message* is received by an actor. Incoming messages trigger transitions associated with the actor's finite state machine. Actions may be associated with transitions, as well as entry and exit

points of a state. The sending of messages to other actors is initiated by an action. The finite state machine behaviour model imposes that only one transition at a time can be executed by each actor. As a consequence, a *run-to-completion* paradigm applies to state transitions. This implies that the processing of a message cannot be preempted by the arrival of new (higher priority) message for the same actor. However, as explained later, in a multi-threaded implementation, the processing may be preempted by other higher priority threads.

ROOM supports the notion of a *composite* state, which can be decomposed into substates. Decomposition of a state into substates can be taken up to an arbitrary level in a recursive manner. The current state of such a system is defined by a nested chain of states called a *state context*. The behaviour is said to be simultaneously "in" all of these states. Transitions on the innermost current state take precedence over equivalent transitions in higher scopes. An event for which no transition is triggered at all levels of the state hierarchy is discarded, unless it is explicitly deferred.

ROOM also provides the concept of a layered architecture. A layer provides a set of services to the entities in the layer above. The linkage between layers is done at discrete contact points which are called *service access points* (SAPs) in the upper layer which uses the services, and *service provision points* (SPPs) at the layer providing the services. Each service access point is connected to a service provision point in the layer below (there can be a many to one mapping), and the end points of each such connection must have matching service points.

The bottom layer in ROOM models is provided by the ROOM virtual machine, which provides, among other things, a *communications service* and a *timing service*. The communications service provides the services to establish and manage connections between ROOM actors. The timing service may be used to set and cancel timers, both one-shot and periodic. The ROOM virtual machine is also responsible for interfacing to other external (non-ROOM)

environments such as specialized hardware or other software components and systems.

ROOM run-time systems provide an implementation of the ROOM virtual machine, and are responsible for providing the mechanisms that support the ROOM paradigm as well as the services needed by ROOM models. The ObjecTime toolset is a CASE tool that provides a fully integrated development environment to support the ROOM methodology, with features such as graphical and textual editing for actor construction and C++ code generation from the model. The ObjecTime toolset includes a micro run-time system microRTS, which is linked with the application code to provide a standalone executable that may be run on either a workstation (emulation) environment, or on a target environment with an underlying real-time operating system such as VxWorks, QNX, pSOS, and VRTX.

2.2 Radar System Definition

We will model a radar system which will be based on existing ones with respect to minimum range, maximum range, resolution, sphere revolution speed, etc. A sample of radar systems is summarized in Table1 on the next page.

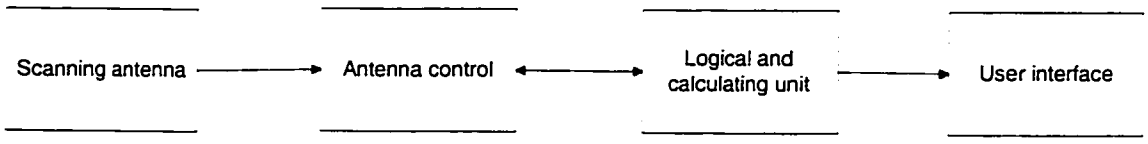

Figure 1. "Real-life" radar system.

Figure 1 shows the basic schema of a general real-life radar system. The Scanning antenna "observes" the field of interest (normally half sphere) by means of sending radio beams and receiving the reflected ones from targets. A beam should be narrow for the purpose of higher resolution, thus we can distinguish between different targets. In order to observe the whole space properly, many beams must be sent in different directions periodically. It is the

job of the Antenna control module to govern the times and directions of beam sending/receiving. We can call this process the '*scanning of the field*'.

Table 1. Typical medium range radars

Model	Range,km	Scan Rate,rpm	Resolution,deg	Type
AN/TPS-61	180	15	2.7	Medium range ground radar
RAMP	171	12	1.5	Air traffic surveillance radar system for Canada
RSR – 1A	288	10	1.25	ATC en-route surveillance
HADR	306	6	1.1	Air defence radar
RAT-31S	270	12	1.5	Air defence, tactical radar
MIR	180	35	1	Steered instrumentation radar
VSTAR	243	20	1	Air defence radar
STAR III	180	15	1	ATC approach control
S511C	180	15	1.5	Approach/ terminal control
Watchman	180	15	1.5	Military surveillance and ATC
TR23K	288	15	1.7	ATC
AN/TPS-70	324	20	1.6	Surveillance radar
SERIES 320	180	6	1.4	Air defence radar
TRS 2215	324	6	1.5	Air defence
ASR-30	252	12	1.25	ATC
AR-I	36	80.4	0.62	Precision approach radar
AN/TPQ-37	29	240	0.6	Weapon locator
AN/TPN-25	27	120	0.5	Precision approach radar
TRS 2310	36	120	1.1	Precision approach radar
AN/APS-116	27	300	2.4	Periscope-detection radar

The information is then fed to logical and calculating unit. Here we must distinguish between the targets. For example, a typical question would be if we observed one target at time t at some location, and another target appeared nearby at time $t + \Delta t$, then are those two observations of the same target, or of two different targets? In order to accurately track targets based on observations of their location, we must estimate their speed and direction of target motion, and then use these estimates to "watch" their movements. We have to continue 'watching' a target, based on new information being supplied over and over by the scanning antenna. When the useful information is obtained (credentials such as location, direction, speed, and, for more complicated systems, even the type of target based on the former), it is ready for the user interface block. This block can be an actual radar screen for human perception, where we can visually display all these target credentials, it can be an interface of some weapon control system, or an interface of an air traffic control computer for further decisions.

2.3 Summary

In this chapter we have discussed ROOM as one of the methodologies used in real-time systems design and looked at its features. We also presented a short summary of various existing radar systems.

3 Simulated Radar System

In this chapter we will discuss the design of the simulated radar system. We choose to simulate a medium range radar with high demanding technical characteristics. Typically, medium range radars do not use very high scan frequencies since distant targets cannot change their azimuth and elevation significantly in a short time interval. The radar characteristics we have chosen increase the scan frequency and resolution as compared to medium range radars shown in Table1. This means that the radar will be able to track targets with more precision and to closer distances, but imposes more load on the system. Here are the characteristics of our ground based medium range radar:

Effective range	10km – 300km
Resolution	0.5 degree
Distance resolution	1 m
Azimuth angle	0 – 360 degrees
Elevation	0 – 90 degrees
Antenna total scan period	0.1 sec.

We assume the radar will have an electronic beam antenna, which will see the outer environment as a half-sphere. The output of the antenna is in the form of a two-dimensional field, where the two dimensions represent azimuth and elevation angles, and the values represent distance to the target. The output will be fed into intelligence blocks of radar, the primary goal of which is to detect new targets and constantly predict and track target motion. In a real life system, achievement of this goal would be enough to supply appropriate information for weapon-control systems, graphically plot target trajectory and speed vector, etc.

To consider the task for our work, we built a mathematical model of a real radar system, which we call the Radar Simulator System. From here on, the

term 'Radar' will denote our Radar Simulator System. The basic scheme of the Radar is shown in Figure 2.

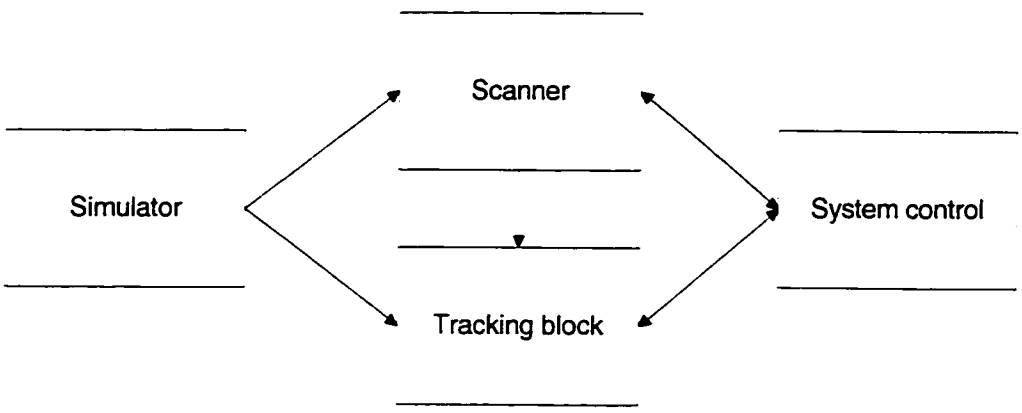

Figure 2. Radar System.

In this approach, we focus our attention only on main intelligence blocks – the new target identifier ('Scanner') and the target tracking block. Everything which goes before this area – antenna, antenna output and the working environment itself, we consider as one block, the Simulator. We also don't worry about where the output of the radar goes – such as a graphical display, or special output for weapon control or other systems. In our case the Scanner, the Tracking block and the System control would correspond to a Logical and Calculation unit in a real life radar. This approach would let us basically look into one area: algorithms, principles and policies of successful target identification and tracking.

In Section 3.1 below, we present the structural and behavioural models of the implemented system. These models were developed using the ObjecTime Developer Toolset, and augmented with code-segments on state machine models to complete the design. The automatic code-generation facilities of ObjecTime Developer were used to produce executable code for the Solaris platform. We fully utilized ObjecTime ROOM-based architecture for building

objects, or 'actors', design communication between them, defining timer services, specify communication protocols and design state machines with the transitions (transactions in ObjecTime) fired according to appropriate communication messages (signals in ObjecTime), bearing certain message priorities. Note that for state machine transitions, ObjecTime specific term 'transaction' is used throughout the text. Thus, all the code for the above-mentioned infrastructures was generated by ObjecTime for us, and we have only to "fill out" the code for the state machine transactions (also, for state-entry and state-exit points at some cases), which serves as actual 'payload' of our application. The only exception is, as we will see later, the Relative Timer needed for certain design. It was built explicitly as an ObjecTime actor.

In Section 3.2, we show how the targets and their motion are modeled in the implemented system. Section 3.3 gives algorithmic aspects of target tracking based on the target models described in Section 3.2. Finally, Section 3.4 gives an overview of how we model the value or importance of targets.

3.1 Structural and Behavioural Models

Figure 3 shows the top-level structure of the application – the main system actor. The system consists of two sub-systems – simulator and tracking block. Each of the two sub-systems is implemented using ROOM actors. The main system communicates with the simulator subsystem using the "SimControl" port, and with the radar subsystem using the "RadarControl" port. Both these ports are of the type "sysControl", which allows basic subsystem control messages.

The behaviour of the main system actor is straightforward – it mainly serves to start the system simulation and then perform post-simulation actions when the simulation is done. When the system starts, it transitions to the state "simMode" – initialization steps are performed as part of this transition, and the system simulation is started. When the simulation is finished, the system executes post-simulation action during the "stopping" transition before exiting.

During initialization, the main system reads an input Scenario file which defines the behaviour for all the targets in the environment for the duration of the experiment. The output of the system is the "Observation" file, in which the radar system's perspective of the behaviour for all the targets in the environment for the duration of the experiment is captured for post-processing. The basic format of the input and output files is in the form of a table, where rows are time steps and columns are target ID's. The table contains information about each target at a given moment of time, such as azimuth and elevation angles and direct distance to the radar station. In addition, the output contains some information about target importance (how 'critical' the target is at the moment) for analyzing experiment results.

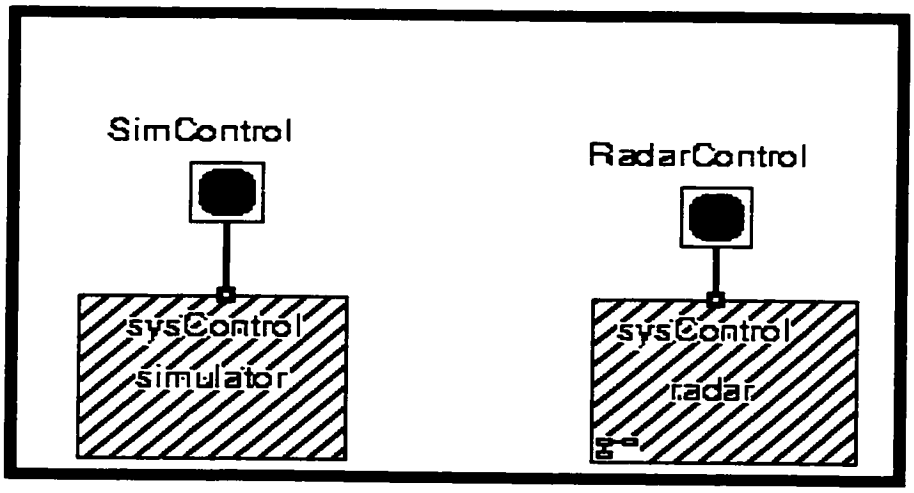

Figure 3. Structure of the Main System Actor

3.1.1 Simulator Sub-System

The simulator sub-system is implemented using the "Simulator" actor. It simulates the periodic output of the antenna. It updates a two-dimensional global array according to the scenario file every 0.1 sec. Rows and columns of the array represent azimuth and elevation angles respectively, with the step of 0.4 degree while the values are distances to targets. Value 0 shows absence of a target in a particular cell. The second function of the "Simulator" actor is to stop the simulation when the end of the scenario is reached and to report to the "Main System" via the communication port. The "Simulator" is connected only to the "Main System" by the communication port "sysControl". The behaviour of "Simulator" actor is shown in Figure 4.

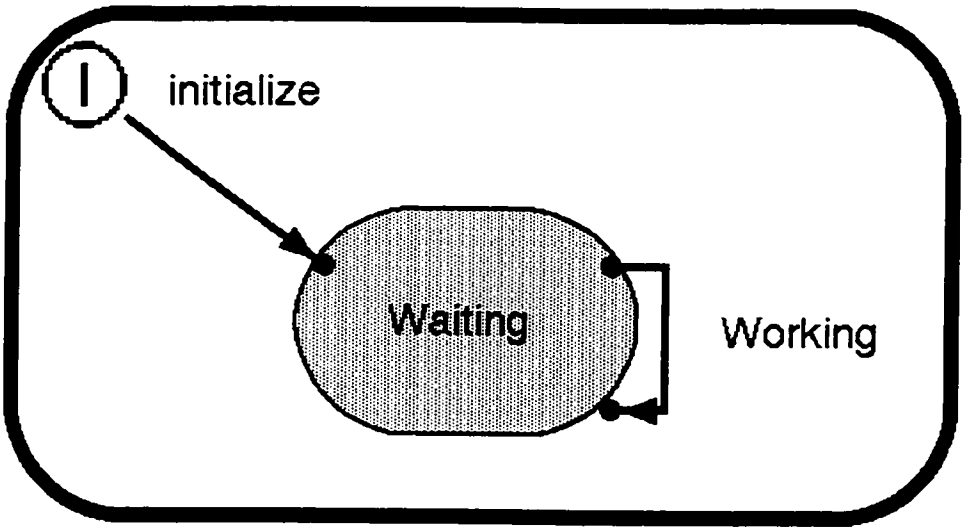

Figure 4. State machine of Simulator actor

After initialization, the "Simulator" enters its only state called "Waiting". Each time, upon receiving system alarm timeout, transaction "Working" is triggered. During this transaction, a two-dimensional array is simply updated according to the current step of the scenario.

3.1.2 Radar Sub-System

The Radar sub-system – implemented by the "Radar" actor – is composed of two actors: "Scan" and "Track", as shown in Figure 5.

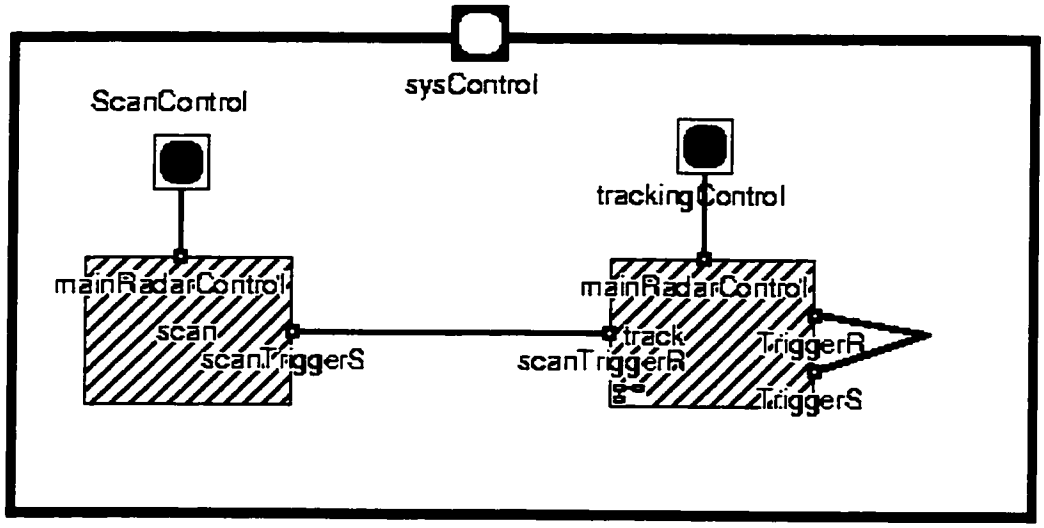

Figure 5. Structure of Radar actor

The purpose of the "Scan" actor is to identify newly appeared targets, while the purpose of the "Track" actor is to track known targets. Both actors are connected to their parent actor ("Radar") by communication ports "scanControl " and "trackingControl" for general control purposes. They also are connected to each other by "scanTriggerS"/"scanTriggerR" to pass information about newly appeared targets from "Scan" to "Track". "Track" actor has also two interconnected trigger ports "TriggerS" and "TriggerR" (*send* and *receive*). They are used to trigger certain actions inside the Track actor, by sending appropriate messages to itself. The Radar actor behaviour is shown in Figure 6.

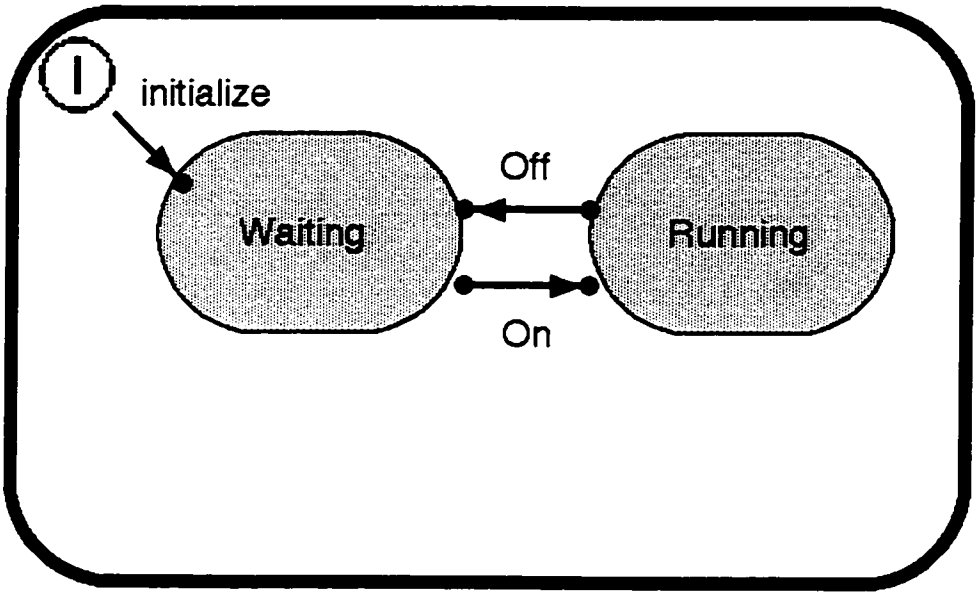

Figure 6: State machine of Radar actor

Upon initialization, the "Radar" is in the "Waiting" state for the purpose of synchronization. When a message (as a signal to trigger "On" transaction) is received from the parent actor ("Main System"), the "Radar" switches to the "Running" state. During the "On" transaction, the "Radar" sends appropriate start messages to its internal actors ("Scan" and "Track"). When the simulation is finished, the appropriate message comes from the "Main System" actor, and the "Radar" switches back to the "Waiting" state. During the "Off" transaction, the "Radar" sends stop messages to its internal actors.

Scan Actor

The purpose of the "Scan" actor is to identify new targets. Within fixed time intervals, it scans through a two-dimensional array of antenna input. Upon any target detection, the "Scan" actor looks at the database of known targets, and, if no match is found, reports the given target as a new one to the "Track" actor. The "Scan" actor is connected to the "Radar" actor and to the "Track" actor by the communication ports "mainRadarControl" and "scanTriggerS" respectively (c.f. Figure 5). The behaviour of the "Scan" actor is shown in Figure 7.

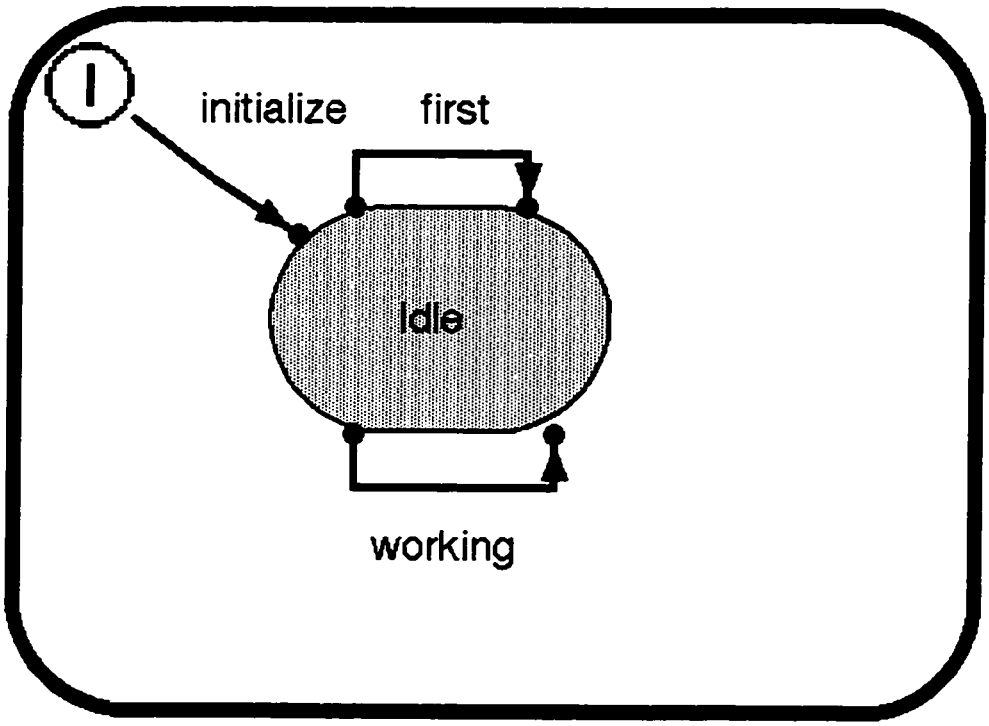

Figure 7. State machine of Scan actor

Upon initialization, the "Scan" actor waits for the start message from the "Radar" actor, and then fires "First" transaction. A timer for the periodic triggering of the "Working" transaction is set during execution of the "First" transaction. Basically, algorithms for "Working" and "First" transactions are the same, however, specificity of design of the first scanning requires two transactions to be present.

Track Actor

The "Track" actor manages tracking of known targets. It contains replicated "Target" actors as shown in Figure 8. Each of the "Target" actor contained in the "Track" actor represents one target that is being tracked. The "Track" actor also contains a "Timer" actor that provides timing services.

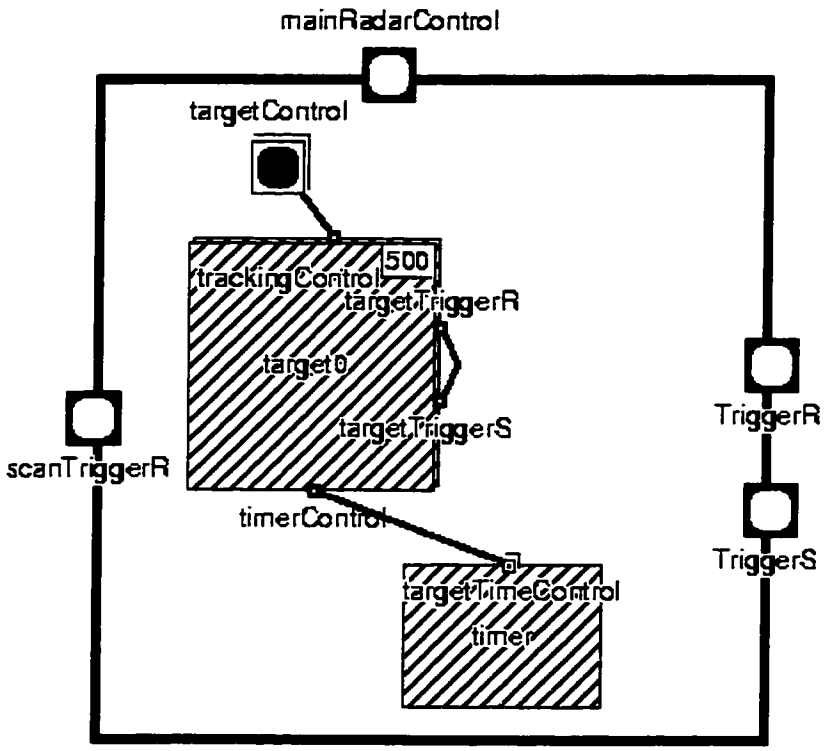

Figure 8. Structure of Track actor

The behaviour of "Track" actor is shown in Figure 9. The "Track" actor receives a message from the "Scan" actor with the observed credentials of a new target, picks up available replicated "Target" actor (being dormant up to this moment) and assigns the target to it. After initialization, upon receiving a message from the parent actor ("Radar"), the "Track" actor goes to tracking state. There is an 'embedded state' inside tracking state, which has only one transaction, which leads to itself: "ProcessUT". When the "Scan" detects a new target and "Track" actor receives an appropriate message, transaction "ProcessUT" is fired. During this transaction, as it has been just said, one available "Target" replicated actor is chosen and sent the message to begin to 'work out' the target.

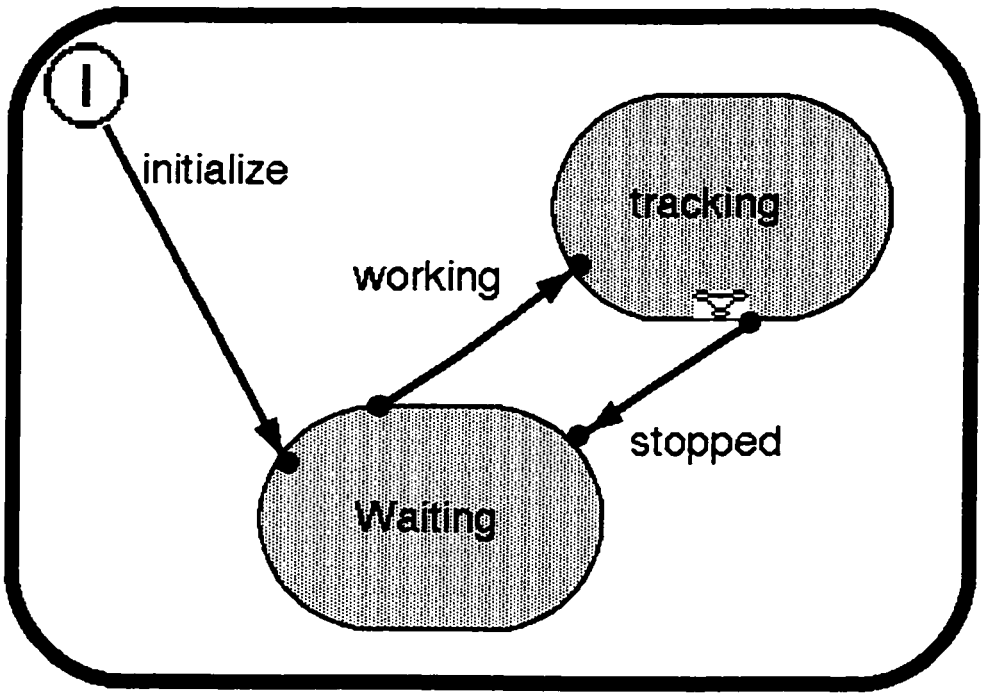

Figure 9. State machine of Track actor

Target Actor

The "Target" actor is a replicated actor. The number of "Target" actors corresponds to the absolute total limit of targets the System can handle. The "Target" actor is a plain actor, which has one communication port to receive messages from the parent actor ("Track") via port "trackingControl" (see Figure 8). It also has two trigger ports "targetTriggerS" and "targetTriggerR", connected to each other. They are used to asynchronously trigger internal "Target" transactions. The behaviour of each replicated "Target" actor is shown in Figure 10.

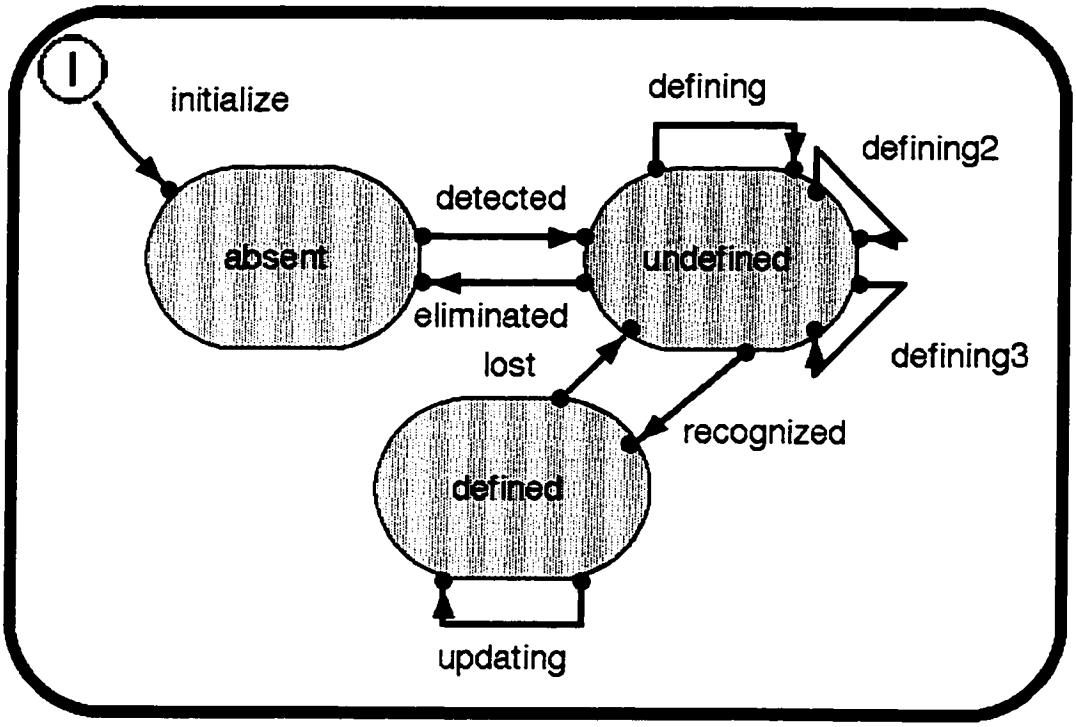

Figure 10. State machine of Target actor

After initialization, the "Target" is idle (or dormant) in the state "absent", until an appropriate message is received by the given replicated "Target" actor. Then transaction "detected" to state "undefined" occurs. This transaction is the first step of "undefined" target processing algorithm. The transactions "defining", "defining2" and "defining3" are the three steps of this algorithm. They execute within an appropriate time period and may be repeated as necessary, in order to complete the algorithm.

If the "undefined" state transactions are completed successfully, the state changes to "defined" via "recognized" transaction. Otherwise, the target is considered to be lost and returns to "absent" state via "eliminated" transaction.

In the "defined" state, tracking occurs periodically within "updating" transaction, which uses a different algorithm to track a target, the motion of which can be predicted. This algorithm takes into account previously known speed and

acceleration vectors. If the target is lost during routine tracking, the state returns to Undefined via transaction "lost".

Note that there are two active groups a target may belong to: "Undefined Targets" (UT) and "Defined Targets" (DT). The "Undefined Target" is the one for which only the recent location is known. "Defined Target" is the one for which the last speed and acceleration vectors are known.

Timer Actor

In case relative time is needed rather than strict real time (which basic ObjecTime Timer actor maintains), a "Timer" actor provides auxiliary timing functions for the "Radar" actor. The service provided by the "Timer" is somewhat similar to the *informEvery* standard ObjecTime service. The "Timer" is a plain actor, which is connected only to the "Target" actors by the replicated communication port "targetTimeControl" (see Figure 8). The behaviour of "Timer" actor is shown in Figure 11.

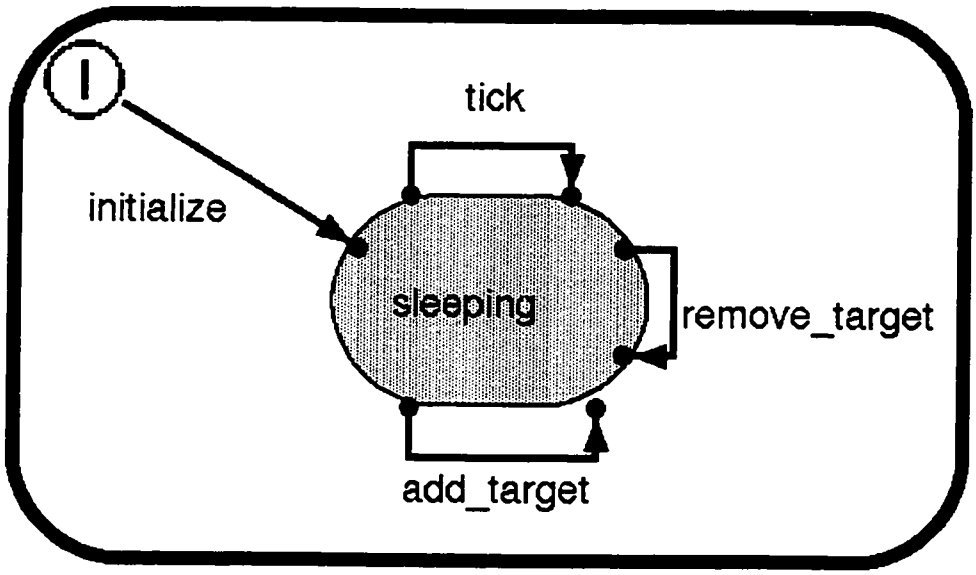

Figure 11. State machine of Timer actor

The "Track" actor can add a target to the list of targets that the "Timer" actor 'monitors'. At this point transaction "add_target" is invoked. For the reverse purpose, the transaction "remove_target" exists.

At each "tick" transaction, which is invoked by the regular system timer according to 'our timer' granularity, the "Timer" updates the global value for the general system load and verifies if there are any expired timeouts for targets on the list. Such targets are immediately notified. Further they start performing their calculations by themselves. For a certain type of model implementation, "Timer" calculates a new invocation period for the recently awaken object. This calculation is done in accordance to the target importance value and general system loading at the moment (see "period-driven approach" in 4.3.3).

3.2 Modeling of Target Objects

The main objective of the target-tracking radar system is to track the target objects flying in the radar's field of view. The "simulator" sub-system in the ROOM model generates the output of the radar antenna that is then used by the target-tracking sub-system. The generation of the antenna output and the tracking algorithms are both dependent on how the targets are modeled.

The antenna output, i.e., the output from the simulator system, is a series of discrete observations (one for each "scan") about where the objects are identified in its field of view. These target positions are then used as the basis for target tracking. Successive observations are used to obtain a target's velocity and acceleration. Based on a target's current position, velocity, and acceleration, a target's future position can be predicted for the next observation step. The predicted position can then be compared with actual observed positions (from the antenna's output) for tracking purposes.

The antenna output itself is driven by pre-computed trajectories of simulated targets as read from an input "scenario" file. This scenario file is generated by an auxiliary scenario generator program used in our experiments, as detailed later in this thesis.

During the stages of target identification and tracking, we mostly use methods of three-dimensional analytic geometry. We first review some basic definitions that are used in the following discussions.

3.2.1 Basic Definitions

The following basic definitions are used to represent the position and motion of the moving targets.

Definition 1.

A vector is the value with a direction. In our case direction can be denoted by angles to x, y and z axes. Any such vector can then be represented as $\vec{R} = \vec{i} * P_x + \vec{j} * P_y + \vec{k} * P_z$ (where $\vec{i}$, $\vec{j}$ and $\vec{k}$ are unit vectors along the x, y, and z axes respectively, and P_x, P_y and P_z are the projections of the vector on the corresponding axis).

Definition 2.

A value with no direction is a scalar.

Definition 3.

The modulus or length |R| of a vector $\vec{R} = \vec{i} * P_x + \vec{j} * P_y + \vec{k} * P_z$ is given by:

$$|R| = \sqrt{P_x^2 + P_y^2 + P_z^2} .$$

Definition 4.

The Azimuth of a vector $\vec{R} = \vec{i} * P_x + \vec{j} * P_y + \vec{k} * P_z$ is given by $arccos\left[P_x / |R| \right]$ and the elevation of a vector is given by

$$arcsin\left[P_z / |R| \right] \qquad (3\text{-}1)$$

3.2.2 Target Object Definitions

In this Section, we use the above definitions to describe a target's position, velocity, and acceleration within the context of the radar system. Also, we show how given a target's current position, velocity and acceleration, its future position can be predicted for the next observation step.

Definition 5. Target Position

The position of a target can be represented by a Radius-vector $\vec{R} = \vec{i} * P_x + \vec{j} * P_y + \vec{k} * P_z$ that originates from the origin of the current coordinate system. However, for target tracking and identification purposes, it is more convenient to represent target's position by its modulus, or distance from the origin, *azimuth*, and *elevation.*

Definition 6. Target Velocity

The velocity of a moving target V is a vector such that

$$\vec{V}_i = \frac{\left(\vec{R}_{2i} - \vec{R}_{1i}\right)}{\Delta t} \tag{3-2}$$

where R_1 and R_2 are the current and immediately preceding radius-vectors of the target respectively; Δt is a time interval between the two observations of the target position.

Definition 7. Target Acceleration

The acceleration of a moving target is a vector such that

$$\vec{A} = \frac{\left(\vec{V}_{2i} - \vec{V}_{1i}\right)}{\Delta t} \tag{3-3}$$

where V_2 and V_1 are the current and last target speed-vectors respectively; Δt is the time interval between two recounts.

Definition 8. Predicted Target Position

Given the current radius vector, velocity vector, and acceleration vector, the position of a target can be predicted. The radius-vector of the target at the predicted point is determined by the formulae

$$\vec{R}_p = \vec{R}_{2i} + \vec{V}_i * \Delta t + \vec{A}_i * \Delta t^2$$

$$\left|R_p\right| = \sqrt{x_i^2 + y_i^2 + z_i^2} \qquad (3\text{-}4)$$

Here *i* and *p* stand for current and predicted states respectively.

3.2.3 Auxiliary Scenario Generator

Strictly speaking, the scenario generator is not a part of the radar system, but it directly prepares input, which is used by the Simulator actor of the system. The Simulator itself represents an antenna, which only fixes a location of the objects in the environment space within fixed time intervals. To model such a scenario, we used an algorithm which allows simulation of any number of moving or fixed objects.

The scenario generator will require certain parameters for each target. At the beginning of the life cycle of a target, the appearance time and coordinates (via azimuth, elevation and distance) must be provided, along with the time interval for current type of motion and initial speed. Then the type of motion (straight, circular or parabolic) for a given interval is provided. Depending on this type, some additional parameters will be required, which are described later in this section in corresponding subsections. After the scenario for a given target for a given interval is generated, the program will require information about whether or not the target still exists (actually detectable). If yes, the program will want to 'know' the next time interval of the target's existence, type of motion, and specific motion parameters. This cycle per target repeats over and over, until the whole length of the experiment is taken, or until target is chosen not to exist anymore for the remainder of the current experiment.

When finished with one target, the system requires the information about whether or not to generate any more targets, and, if yes, it starts the whole process from the beginning for the new target. The system will keep on generating targets, until the parameter *no more targets* is entered, or the maximum possible number of targets (300 in our case) is generated.

Objects Moving in a Straight-Line Trajectory

Here we model the most common targets, which are flying along a straight path. They can be approaching the radar, going away, moving across the field up or down, or following a trajectory that fits in between of these. Steady objects can also be modeled this way as well, by defining speed zero at the beginning and the end of the time interval.

We define such a motion per short time interval, where length is given as an input parameter for scenario generator. Speed-vector V of the target is given by providing angles of V to x-axis, $V_{\alpha x}$, and to z-axis $V_{\alpha z}$. Acceleration A is given by providing |V| at the beginning and the end of the given time interval.

Within every time interval of the scenario, which corresponds to one revolution of the antenna, new coordinates of a target are calculated as:

$$x = x_{old} + \left[|V| * cos(V_{\alpha x}) * cos(V_{\alpha z}) + |A| * cos(A_{\alpha x}) * cos(A_{\alpha z}) * \Delta t\right] * \Delta t \quad (3\text{-}5)$$

$$y = y_{old} + \left[|V| * sin(V_{\alpha x}) * cos(V_{\alpha z}) + |A| * sin(A_{\alpha x}) * cos(A_{\alpha z}) * \Delta t\right] * \Delta t \quad (3\text{-}6)$$

$$z = z_{old} + \left[|V| * sin(V_{\alpha z}) + |A| * sin(V_{\alpha z}) * \Delta t\right] * \Delta t \quad (3\text{-}7)$$

where X_{old} is the previous x coordinate of the target. The distance to the target D at each time point is

$$D = \sqrt{x^2 + y^2 + z^2} \quad (3\text{-}8)$$

Azimuth and elevation are

$$Azimuth = arccos\left(\frac{x}{D}\right) \quad (3\text{-}9)$$

$$Elevation = arcsin\left(\frac{z}{D}\right) \tag{3-10}$$

If at some moment the target is no longer moving along a straight path, different formulae will be applied starting from that moment.

Objects Moving in a Circular Trajectory

Here we model targets which are following a circular path at the constant altitude (for simplicity). They can complete the full circle and go on, or just fly along an arc, depending on the time interval we provide for this motion. The circle centre is chosen arbitrarily, and can be the Z axis (above the radar) as well. Altitude remains the same from the previous time interval, or from coordinates of initial appearance.

Targets that are moving along a circular curve are modeled as follows: given (or calculated from the previous step) initial coordinates x_0, y_0 and z_0, coordinates of the circle centre x_r, y_r, z_r and $|V|$ of the target (either given if this is the first life interval of a target, or known from the previous calculations). For simplicity we assume that $|V|$ is constant, i.e. A = 0. Then radius of the circle trajectory R is

$$R = \sqrt{(x_0 - x_r)^2 + (y_0 - y_r)^2 + (z_0 - z_r)^2} \tag{3-11}$$

Then, the current x and z values are calculated as

$$x = x_0 + |V| * cos(\alpha V_x) * \Delta t \tag{3-12}$$

$$z = z_0 + |V| * sin(\alpha V_z) * \Delta t \tag{3-13}$$

And the current y value is calculated by solving equation (3-11).

Objects Moving in a Parabolic Trajectory

This is the type of motion we use to model targets whose trajectory is not straight, but not as curved to form a circular motion. Here, altitude is assumed to be constant, as with circular motion. For parabolic motion, current x and z values

are calculated by formulae (3-12) and (3-13) respectively, while y value is calculated by solving the equation

$$(y - y_0) = (x - x_0)^2 * Pr \qquad (3\text{-}14)$$

where Pr is a parabola parameter which characterizes its steepness.

3.2.4 Antenna Outputs

After coordinates of all the targets within scenario steps are calculated and converted into azimuth-elevation-distance form, they are ready to be placed in the environment space matrix, or a 'field'. As mentioned earlier, the rows of the matrix correspond to azimuth, and columns correspond to elevation, while values of each cell represent distance in meters.

The numbers of rows and columns are n and m respectively, where

$$n = 360/ar \text{ and } m = 90/er$$

where *ar* and *er* are azimuth and elevation resolutions respectively. According to the analysis results of existing radar systems, we assumed *ar* = *er* = 0.4 of a degree. Thus we have a 900x225 matrix.

The target at every time step of the scenario is placed into the field by the Simulator actor as D[i][j], where D is the value of the field matrix cell, i and j are the row and column index respectively. The matrix D is taken from the last calculation of distance, and

$$i = azimuth/ar$$

$$j = elevation/er$$

Then, the previous location (matrix cell) of the target is cleared by placing the value 0.

3.3 Target Tracking Algorithms

The target tracking algorithms are used to track a target's movement in the radar's field of view. When a new target enters the field, it is detected as part

of "scan" operation, and then the target is in the state "undefined" (see Figure 10). The tracking algorithm then attempts to track the target object over the next few tracking iterations – if it is able to successfully track the object, the target object is moved to the state "defined." Different processing steps are applied for targets that are undefined and defined, and these are outlined below.

3.3.1 Undefined Target Processing

Upon detection of a new target, a chosen "Target" actor enters the state of undefined target processing and algorithms, described in this section, are performed. General scheme of "undefined target" processing is given on Figure 12.

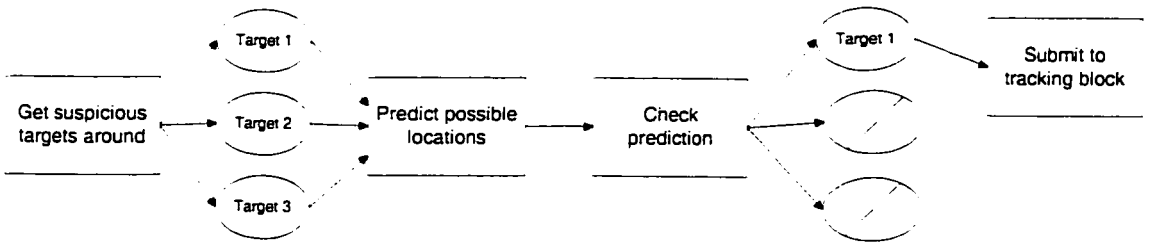

Figure 12. Undefined Target Processing

Assuming that a target moves perpendicularly to the radius vector with speed of 3000 km/h, the target cannot move further than 0.4 degree angle in approximately 0.3 sec. Thus, 0.3 sec serves as the time-interval that is used in successive processing steps.

We can argue that between two such adjacent observations, the target can be in one of the following positions:

- previously observed position [i, j] of matrix, or
- eight positions adjacent to the point [i, j].

This property is taken as a base for the undefined target processing algorithm.

Each step of the algorithm is performed within optimal observation interval as follows:

1. Initial target coordinates azimuth1, elevation1, distance1 are received from the Scan actor.
2. Eight adjacent locations and the previous location are observed to identify possibilities of target movement. For each hit, estimation of a possible speed vector is made based on formula (3-2), as shown in Figure 13.

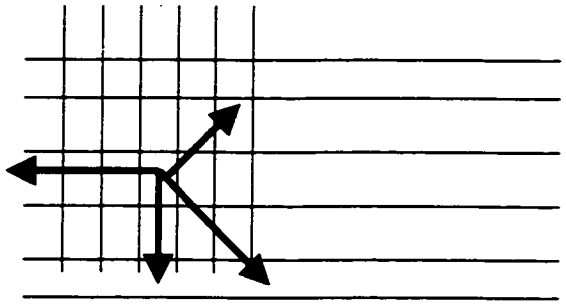

Figure 13. Possible layout of speed vectors.

If only one possibility for the speed vector is left, then it is assumed to be a true speed vector for this target.

Each potential speed vector is calculated according to the following formulae, where indices 1 and 2 represent the previous and current observation per potential target respectively:

$$x_1 = D_1 * cos(\ azimuth_1\ ) * cos(\ elevation_1\ )$$

$$y_1 = D_1 * sin(\ azimuth_1\ ) * cos(\ elevation_1\ )$$

$$z_1 = D_1 * sin(\ azimuth_1\ )$$

$$x_2 = D_2 * cos(\ azimuth_2\ ) * cos(\ elevation_2\ )$$

$$y_2 = D_2 * sin(\ azimuth_2\ ) * cos(\ elevation_2\ )$$

$$z_2 = D_2 * sin(\ azimuth_2\ ) \tag{3-15}$$

Then, projections of a possible speed vector are:

$$V_x = \frac{x_2 - x_1}{\Delta t}$$

$$V_y = \frac{y_2 - y_1}{\Delta t}$$

$$V_z = \frac{z_2 - z_1}{\Delta t}$$

$$|V| = \sqrt{(\ V_x^{\ 2} + V_y^{\ 2} + V_z^{\ 2}\ )} \tag{3-16}$$

Now potential locations of the target can be predicted. For each predicted location, coordinates are calculated as follows:

$$x_3 = x_2 + V_x * \Delta t + A_x * \Delta t^2$$

$$y_3 = y_2 + V_y * \Delta t + A_y * \Delta t^2$$

$$z_3 = z_2 + V_z * \Delta t + A_x * \Delta t^2$$

$$D_3 = \sqrt{x_3^{\ 2} + y_3^{\ 2} + z_3^{\ 2}}$$

$$Azimuth_3 = arccos\left(\frac{x_3}{D_3}\right)$$

$$Elevation_3 = arcsin\left(\frac{z_3}{D_3}\right) \tag{3-17}$$

Upon the next observation, each predicted location is verified, with the error margin taken into consideration. If one of the predictions comes true, the speed vector of the target is refined according to the last observation using formulae (3-16), and control is switched to "Defined Target" processing state. If more than one or none of the predictions come true (due to miscalculation,

incorrect or late observation, irregular target behaviour, etc.), the target identification is considered unsuccessful and "Target" actor is switched to state "absent".

Trying to keep on identifying the target in the latter case will not very likely lead to positive results while it will always consume a large amount of processing resources. At the same time, a target which has been dropped from processing will be again identified as new by "Scan" actor during its next iteration and will be scheduled for identification again.

3.3.2 Defined Target Processing

This is a basic tracking algorithm for a target for which the last speed and acceleration vectors are always known. Coordinates of the predicted location are calculated according to formulae:

$$x_2 = x_1 + V_x * \Delta t + A_x * \Delta t^2$$

$$y_2 = y_1 + V_y * \Delta t + A_y * \Delta t^2$$

$$z_2 = z_1 + V_z * \Delta t + A_z * \Delta t^2$$

$$D_2 = \sqrt{x_2^2 + y_2^2 + z_2^2}$$

$$Azimuth_2 = \arccos\left(\frac{x_2}{D_2}\right)$$

$$Elevation_2 = \arcsin\left(\frac{z_2}{D_2}\right) \qquad (3\text{-}18)$$

where indices 1 and 2 represent the last observation and the current prediction respectively, and Δt is the time interval between last and current calculations.

Knowing the predicted coordinates, we verify them within the field matrix, taking into account possible error margin. In case of successful verification, we

obtain x_3, y_3 and z_3, which are the actual target coordinates at the current moment of time. Now, the actual current speed vector is calculated as:

$$V_{x2} = (x_3 - x_1) * \Delta t$$

$$V_{y2} = (y_3 - y_1) * \Delta t$$

$$V_{z2} = (z_3 - z_1) * \Delta t$$

$$|V_2| = \sqrt{V_{x2}^2 + V_{y2}^2 + V_{z2}^2} \qquad (3\text{-}19)$$

New acceleration is calculated as follows:

$$A_{x2} = \frac{V_{x2} - V_x}{\Delta t}$$

$$A_{y2} = \frac{V_{y2} - V_y}{\Delta t}$$

$$A_{z2} = \frac{V_{z2} - V_z}{\Delta t}$$

$$|A_2| = \sqrt{A_{x2}^2 + A_{y2}^2 + A_{z2}^2} \qquad (3\text{-}20)$$

Values x_3, y_3, z_3, V_2 and A_2 will be used in the next tracking iteration as X_1, y_1,z_1,V and A respectively.

In case of unsuccessful verification, the control is switched to "Undefined target processing" state, for which last known x_1, y and z_1 are used as an input.

3.4 Summary

In this chapter, the structural and behavioural model of our radar system was presented and it was shown how it can be built under ObjecTime. Principles and algorithms of target modeling were explained and antenna output format was presented. Finally, algorithms used for target tracking and new target acquisition were described.

4 Overload Handling

One of the main problems studied in this thesis is the behaviour of the system as the input load on the system increases. In the case of the target-tracking radar system, the load of the system dynamically varies as the number of targets in the radar field increases or decreases. Each target in the field needs to be tracked and the tracking algorithms impose a non-trivial computational load on the system.

Designers of real-time systems must obviously worry about the system performance under overload since correctness of some of the algorithms depends on being able to execute them within their time constraints. The tracking algorithms described in Chapter 3 were developed on the assumption that the algorithm could be successfully executed in every processing step.

What happens with the target if the tracking algorithm is not executed in one processing step? The result may be that the target is lost from the perspective of the tracking system if the critical credentials such as speed vector, direction, etc. are rapidly changing. The probability of losing a target completely for our tracking algorithm will increase if this happens repeatedly. The net result is that under overload, the target tracking algorithms may not be able to successfully track targets.

4.1 Degradation Requirements

We have identified that under overload the system will not be able to successfully track all targets. If the performance of the system will inevitably degrade under overload, then how should it degrade? A simple approach is to not specify any requirements – thus allowing arbitrary degradation of system performance. Alternatively, one can conclude that no degradation is acceptable, and any degradation is equivalent to system failure.

More commonly, system designers would like the system performance to degrade in some graceful manner depending on the needs of the system and the characteristics of the input load. The approach taken here is to assume that

certain targets will be more critical or important than others, and that as the system performance degrades, it should give higher preference to more important targets.

There are multiple reasons why we should discriminate between different targets. More specifically, it is beneficial to give preferential treatment to targets that are nearer to the radar or are moving fast for the following reasons.

1. Generally targets that are closer and especially those that are approaching quickly tend to be more important. For example, in an Air Traffic Control system a plane on a final approach for landing is clearly more important than a plane flying in the distant radar field.
2. Some targets will more likely continue to be successfully tracked even if the processing step is missed occasionally. This will be generally true for targets that are moving slower or are far away (since the angle does not change rapidly) from the radar.
3. Also, targets that are slow moving or far away can afford more recovery time if they are lost for some time.

While it is important to give preference to some targets over others, it is also important to realize that a target's importance may change dynamically. In addition, there are targets in "undefined" state (i.e., it is newly found, but its speed vector and direction are unknown). We will assume that these targets are given higher priority than defined targets to allow the system to move them into a defined state where their relative importance can be evaluated.

4.2 Importance Function

In order to analyze performance and achieve its improvement, we must first distinguish between targets which are more critical (more important) and those which are not. The measure we are going to use will be called the

'Importance Function'. The importance function is used to mathematically define the relative importance of a target that is being tracked by the radar system.

We take into account four important properties of target motion to define how critical it is, relative to other targets.

1. Proximity: Closer targets are obviously more important than farther ones.
2. Approaching vs. going away: Targets approaching the radar are more important than those which are going away.
3. Angular speed: Targets which are going across the field are more important than those which are going straight to (from) the radar, since former ones are more difficult to track due to rapidly changing observation angles (Azimuth and Elevation).
4. Actual target speed: Faster moving targets are more important than slower ones. They require more 'attention' and are more difficult to track.

However, we must recognise that a choice of Importance function in real life will depend on the actual purpose of the radar system. For example, Importance function for a radar used in air traffic control systems will differ significantly from the one used in weapon control systems. In our case, we derived a very broad and generic Importance function, which is to help us to experiment with the performance improvement of our ROOM-modeled system.

The Importance value per each target will be calculated the first time when the transfer occurs from "Undefined Target Processing" state to "Defined Target Processing" state, and then, every time a routine target tracking transaction is performed.

This function is defined as follows:

if $|azimuth_2 - azimuth_1| >= |elevation_2 - elevation_1|$

$$f_i = \frac{|azimuth_2 - azimuth_1| * V_x}{|x_2 - x_1| * resolution} * \frac{D_1}{D_2}$$

else

$$f_i = \frac{|elevation_2 - elevation_1| * V_z}{|z_2 - z_1| * resolution} * \frac{D_1}{D_2}$$

(4-1)

where V_x and V_z are the current projections of the speed vector, indices 2 and 1 denote current and previous observation respectively; i.e., change of target's azimuth is more significant than change of elevation,

Formulae (4-1) are half-empirical. They are derived according to the following considerations:

Values $|x_2 - x_1| / V_x$ and $|z_2 - z_1| / V_z$ are times at which the target moves from one location to another in horizontal and vertical planes respectively. Then, $\vec{V}_{azimuth} = \frac{|azimuth_2 - azimuth_1| * V_x}{|x_2 - x_1|}$, and $\vec{V}_{elevation} = \frac{|elevation_2 - elevation_1| * V_z}{|z_2 - z_1|}$ are the angular speeds of change of azimuth and elevation of target respectively.

Resolution divided over $V_{azimuth}$ and $V_{elevation}$ gives a period T_i, for which a target will pass the angle equal to the resolution within the appropriate plane. If a period of target tracking is bigger than T_i, the target might be lost. Thus, frequency of routine target tracking must be not less than $f_i = 1/T_i$. At the same time, this formula implicitly takes care of general target speed property, since $V_{azimuth}$ and $V_{elevation}$ are directly proportional to the speed.

So far, we obtained formulae (4-1) without term D_1/D_2. But the fact that the target is either approaching the radar, or the target is going away from it, must

also affect the value of importance function. Then, we multiply f_i by D_1/D_2 to reflect this aspect.

Now we must consider the fact that the closer the target is, the more important it is, since it is easier to lose it. But this property would be already reflected in angle speeds of azimuth and elevation change, thus, it needs no special consideration in formulae (4-1).

4.3 Scheduling under Overload

In the previous section we showed how the importance of the different targets being tracked can be characterized. The question then is how we control the scheduling of the processing for different targets with dynamically changing importance, so that higher importance targets are tracked in preference to lower importance targets.

Since the targets are tracked periodically, the basic support in ObjecTime to achieve this is to set up a periodic timer that will periodically queue up a timeout message for the target tracking actor to process. The timer service is part of ObjecTime's run-time library. As part of handling the timeout message, the target tracking algorithm is executed.

The timer service implementation in ObjecTime implements an overload control mechanism to avoid wasting memory resources as queued up timeout messages for a recipient that may not get a chance to free (dequeue) the messages and free the memory. Thus, the timer service only queues up one timeout message for any particular periodic timer – if the timer expires before the previously queued timeout message is received then the new timeout message is not queued (in effect, it is dropped).

In the context of the target-tracking system this would mean that when a target actor is overloaded (i.e., it has not been able to retrieve a timeout message for processing) then no new timeout messages will arrive. In effect, this means that the target actor will skip the execution of the tracking algorithm for one or

more intervals. This will, in turn, result in the likelihood of the target tracking to fail.

ObjecTime provides additional mechanisms to prioritize different processing within an ObjecTime design. First, it allows the user to map different actors to different tasks of the underlying operating system and to assign them priorities. During initialization, the different threads are created and priorities assigned to them. Second, ObjecTime allows the messages to be prioritized. Within a single thread, messages are processed in priority order.

The challenge for our target-tracking system is that the priorities for targets are dynamically changing. If each target is implemented as a separate task, this would require frequent priority change operation for each task which can incur significant overheads. Instead, we choose more lightweight overload handling policies that incur less overhead.

We implemented three different overload handling policies. The same ROOM model was used for all the three policies although some implementation differences arise. In the next chapter we provide experimental results on how well each of these policies performed under varying load conditions.

- **Base System**: In the base system, we implemented no prioritization between different targets.
- **Priority driven approach**: This approach used the message priorities as a mechanism to give higher priorities to more important targets.
- **Execution period based approach**: All the targets have the same priorities, but are assigned different periods of routine tracking, corresponding to their importance functions. Thus, the more critical a target is, the more "attention", or simply more frequent processing it needs.

4.3.1 Base System

This system employs no operations and algorithms other than these described in section 3.2. The frequency of routine tracking transaction is set to be constant, according to the value required to successfully track the most dangerous target. This message is triggered by ObjecTime timer service as described above. The priorities of all the targets in the Defined Target Processing state are the same.

When the system is overloaded, some *arbitrary* targets will incur missed deadlines. Practically, the Timer actor itself will discover overdue timeouts and drop them. If a target can recover after missed deadline(s), it continues to run in the "Defined Target Processing" state; otherwise it is considered lost and becomes an "Undefined Target".

4.3.2 Priority Driven Approach

Priority driven approach employs Importance value of a target (function (4-1)), to determine the priority level at which a target's tracking algorithm will execute. To do this, we used ObjecTime's message priorities as the mechanism to prioritize the processing. However, ObjecTime's default implementation provides a small number of priority levels only. We recompiled ObjecTime's run-time system to support a large number of priorities to be able to discriminate between different targets.

As in the base system, each target executed at a fixed rate, through the use of ObjecTime's timer service. The importance function was used to calculate the importance of a target and that was then mapped to a priority level. The subsequent timer for the target was then set to send messages at the priority level of the target.

When the system is overloaded, the targets with lower priorities will miss their deadlines first. In this case, as before, the Timer service itself will discover overdue timeouts and drop them. If a target can recover after missed deadline(s),

it continues to run in the state "Defined Target Processing"; otherwise it is considered lost and becomes an "Undefined Target".

4.3.3 Execution Period Based Approach

The execution period based approach is based on a couple of important insights. First, we note that changing priorities dynamically is expensive. The overhead of changing priorities will then diminish the gain of being successfully able to discriminate between different targets. In this approach, we try a lighter weight scheme that does not depend on changing priorities dynamically.

Second, we note that there is no real reason to run the target tracking algorithms at a constant rate. More specifically, targets that are lower in importance can be tracked at a slower rate than targets that are higher in importance.

The basic idea of this approach is to dynamically vary the rates of tracking different targets to achieve two simultaneous objectives: (1) The tracking rates should reflect the importance of the target, and (2) The overall system load should be maintained at a level such that no overload occurs. By avoiding overload from occurring, we do not need to prioritize the messages or tasks in the system, giving a relatively low overhead.

To achieve this, the importance function of a target is mapped to a relative frequency at which the target should be tracked. The actual rate at which the target is tracked depends on the system load – the higher the load, the lower the rate. The implementation of this scheme is driven by a new timer service – Relative Timer Service – implemented as "Relative Timer" actor that was described in Chapter 3.

This actor is driven by the basic ObjecTime timer using constant *InformEvery* service. *InformEvery* serves as synchronization points for "Relative Timer". "Relative Timer" will send wake up signals to the targets requiring service, according to the importance value of each target. Targets must report

importance value to "Relative Timer" each time it changes. The speed at which the "Relative Timer" 'ticks' is determined by General System Overload Value.

The General System Overload Function looks as follows:

$$F = \sum \frac{f_i}{N} \tag{4-2}$$

where N is total number of targets which are not in the "absent" state. Here, we must consider undefined targets as well, to have a more precise picture of system load. They are included in N number and they are assigned constant f_i's. System Overload Function F has a unit of frequency.

The actual period of tracking per target is then given by

$$T_i = \frac{\sum f_i}{N * f_i^2} = \frac{F}{f_i^2} \tag{4-3}$$

Formula (4-3) is important from the perspective of dynamic mode of operation of the whole Radar system. We can consider it to be the average Importance of all the active targets. Expression F/f_i is a non-unit value. It defines the relation between the average system importance and particular target importance, i.e, how many times the average Importance is bigger or smaller than the target's Importance.

Value $1/f_i$ is simply a period at which the target is to have routine tracking transaction. Thus, $T_i = \frac{F}{f_i} * \frac{1}{f_i}$ (which reduces to (4-3)) is the period i-th target will be served depending on the load of the whole system and its own Importance.

The typical behaviour of function (4-3) for different Target Importance values f_i is shown on Figure 14.

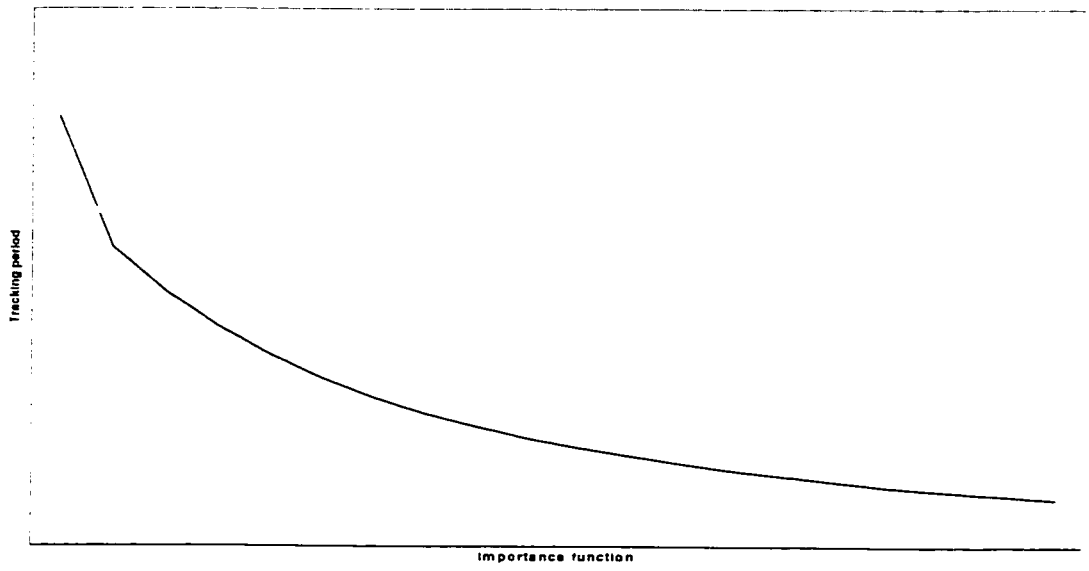

Figure 14. Tracking period for targets of different Importance vs. General System Load.

Note that the variation of a target Tracking period T_i with system load depends highly on the Importance Value of the target f_i. For more important targets, this variation is small and vice versa.

Using the above principles, there is no need for additional control blocks. Values of F (and N) are global and automatically updated by every target, should its Importance Value change, while the Relative Timer simply monitors them to determine how fast it should 'tick' depending on F.

At every synchro-impulse, the Relative timer count is updated as follows:

$$C_n = C_{n-1} + \frac{1}{F} \tag{4-4}$$

Thus, the more loaded the system is, the slower the timer goes. Additionally, the Relative timer behaves like the general ObjectTime Timer, i.e. it drops overdue timeouts and serves only those that are currently due.

4.4 Summary

In this chapter it was explained how our system is to behave under overload conditions. Higher preference was given to more important targets to make the system degrade gracefully. The function responsible for assigning priorities to targets was defined. Finally, we presented three different approaches to target scheduling. In base system we did not differentiate between targets so arbitrary targets incurred missed deadline in overload condition. In the remaining two approaches, higher importance targets were tracked in preference to lower importance ones. The value calculated using our Importance function was used in the priority driven approach to map it to the priority of the message sent by subsequent timer for the target. The execution period based approach took into account target priorities to dynamically vary the rates of tracking different targets as opposed to changing message priorities. There, an effort was made to avoid overload from occurring by slowing the timer when the load increase.

5 Experimental Results

In this chapter we will present the design of an experimental setup to evaluate the different overload handling policies presented in Chapter 4. Also, the results of those experiments are described.

5.1 Experimental Setup

The major components of our experimental setup and their relationships are shown in Figure 15. The same general setup was used for all the three overload handling policies implemented in our radar system. The entire system was implemented on a Sun Sparc 5 workstation running Solaris 2.6 operating system. The real-time scheduling classes were used to provide the real-time characteristics for the radar system.

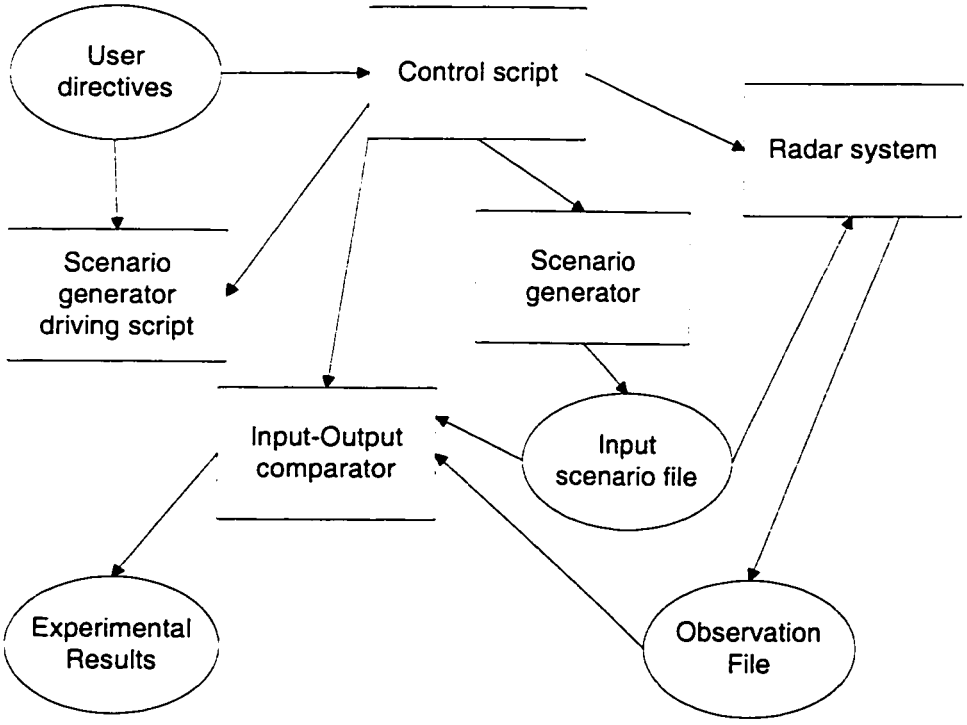

Figure 15. Experimental Setup

The control script was written in Perl. It governs the execution of the whole experiment. It starts the appropriate scripts and/or binary executables at the

appropriate times, assuring synchronization, since all the blocks execute in strictly sequential order (i.e., the latter block starts only upon the total job completion of the former one). The scenario generator script invokes and passes appropriate parameters to the scenario generator, using hard-coded user directives. The scenario generator is a binary executable that produces the scenario file (see Section 3.2.3 for a description). It employs algorithms for scenario generating as described in 3.2.3: "Auxiliary Scenario Generator".

The Radar system is the ROOM/ObjecTime implementation of the Radar system as described in Chapter 3. It takes the Scenario file as an input and produces the Observation file as an output. The output of the radar system is analyzed by the input-output comparator executable that evaluates the performance of the Radar System by comparing the Scenario and Observation files. Comparison results, necessary statistics and calculations are stored in the Experiment Results file.

5.2 Load Scenarios

To evaluate the three overload handling policies, we created different load scenarios. The intent was to observe the effect on system performance as the load on the system was varied. The loads were controlled by varying two parameters. First, we varied the number of targets that were used in the system.

In addition, we also considered the fact that the actual amount of data to be analyzed as well as algorithms for the analysis may differ for various real-life systems. To simulate its effect on the load, we artificially imposed a computational load in the target-tracking part of the algorithms by implementing an idle for-loop that simply used up processor time. This load was characterized as low (approx 0.4ms), medium (approx 10ms), and high (approx 900 ms).

5.3 Experiment Scenario Generation

For all the systems, a similar scenario was used. Scenarios differ only by the maximum number of targets. This scenario utilizes a variety of different targets, flying in different directions, at different speeds.

The scenario generator starts with no targets and then generates new targets until their number reached the maximum number of targets for a given experiment. In total, each experiment runs for 5 minutes. The number of targets was varied from a minimum of 5 to a maximum of 300 in increments of 5.

The targets were generated so that they appeared all around the horizon, from various elevation angles (from 0 to 80 degrees). Every 10^{th} target followed a parabolic trajectory, and every 50^{th} target followed a circular trajectory around the radar. All other targets were generated to fly straight towards the radar in terms of horizontal direction, and a bit above it (1 degree) in terms of vertical direction, so they will eventually fly above the radar and continue their way towards the other side of the space.

The target generation was spaced so that ten targets were generated every second. The initial speeds of the targets were set such that the initial target speed increased in increments of 5m/sec from the first generated target to the last generated with the last target having an initial speed of 400m/sec. All targets had constant acceleration such that their final speed was 100m/sec more than their initial speed. Targets flying by parabolic and circular motion had their linear speeds constant, equal to the initial speed, defined as above.

The initial positions of the targets were varied so that they were generated all around the observation field, with changing elevations (in increments of 2 degrees). Hence during the experiment the targets would fly all over the half–sphere in different directions.

The manner in which the targets were generated resulted in the load on the system increasing as the experiment progressed. This was because (a) the number of targets kept increasing, (b) the speeds of the newly appearing targets was progressively increased and (c) the targets moved closer to the radar. However, by the end of the 5-minute run, some targets were able to actually pass by the radar (and to start going away, or even disappear beyond the observation boundaries), resulting in a lowering of load.

5.4 Assessing System Performance

In order to assess how effective the implemented policies were in ensuring good system performance as the load on the system was varied, we created an objective metric – compound relative performance (P) -- which was calculated according to the following formulae:

$$P = \sum \frac{P_i}{TotalNumberOfSteps} \tag{5-1}$$

$$P_i = \sum \left| R_i real - R_i observed \right| * f_i \tag{5-2}$$

In these formulae, the relative performance for each step of the simulation (this corresponds to the steps in the scenario and observation file) was calculated by summing up a metric that measured how accurately a given target was being tracked. Here, R_i real and R_i observed are the radius-vectors of a given target according to the scenario file and radar output file respectively, and f_i is the importance function of a given target at a given moment. The overall system performance was then obtained by taking the average performance of each step.

The relative performance metric thus gives an indication of how accurately the radar was able to track the targets. The larger the error in tracking, the larger is the value of this metric. In other words, a large value of relative performance indicates poor system performance.

The comparator program generated values of this metric by comparing the output observation file and the input scenario file. Further analysis of the experimental results along with plotting appropriate charts was done using the Microsoft Excel.

5.4.1 Experimental Results

The experimental results are plotted in Figures 16, 17, 18, 19 – one for each of the four calculation load scenarios (no load, low load, medium load,

heavy load). In each plot, the relative performance is plotted on the y-axis and the number of targets is plotted on the x-axis. In each Figure, the performance of each of the three overload handling policies is given making it easy to compare their relative performances. Note that for visual clarity, only Figure 16 uses linear scale for y-axis, while the remaining diagrams use a logarithmic scale.

Figure 16 shows us that if there is no calculation load, there is almost no difference in performance of plain and priority driven systems. However, even here we can see a slightly better performance on period driven system. All three systems behave roughly the same as the number of targets increases. This is likely because with no calculation load the system was fast enough even with the maximum number of targets. Figure 17 shows the performance under the low calculation load scenario. We can see that when the number of targets is relatively small, all three systems have approximately the same level of performance. When the total number of targets approaches 190, the performance of the plain system suddenly becomes much worse due to the "avalanche effect" (i.e. the system starts loosing such a significant number of important targets that while it is trying to recover them, it looses the other critical targets and so on), while for the priority and period driven systems, it still remains at a high level. With total number of targets reaching 225, the performance of the priority driven system gets significantly worse, while the performance of the period driven system is just slightly decreasing. The points at which the sudden degradation in performance occurs are clearly the points where the system hits overload behaviour.

If the calculation load is bigger, then the performance deterioration occurs much earlier, as we can see in Figure 18. Here the performance of the plain system significantly degrades at about 20 targets and the performance of the priority driven system starts sharply decreasing at 45-target level.

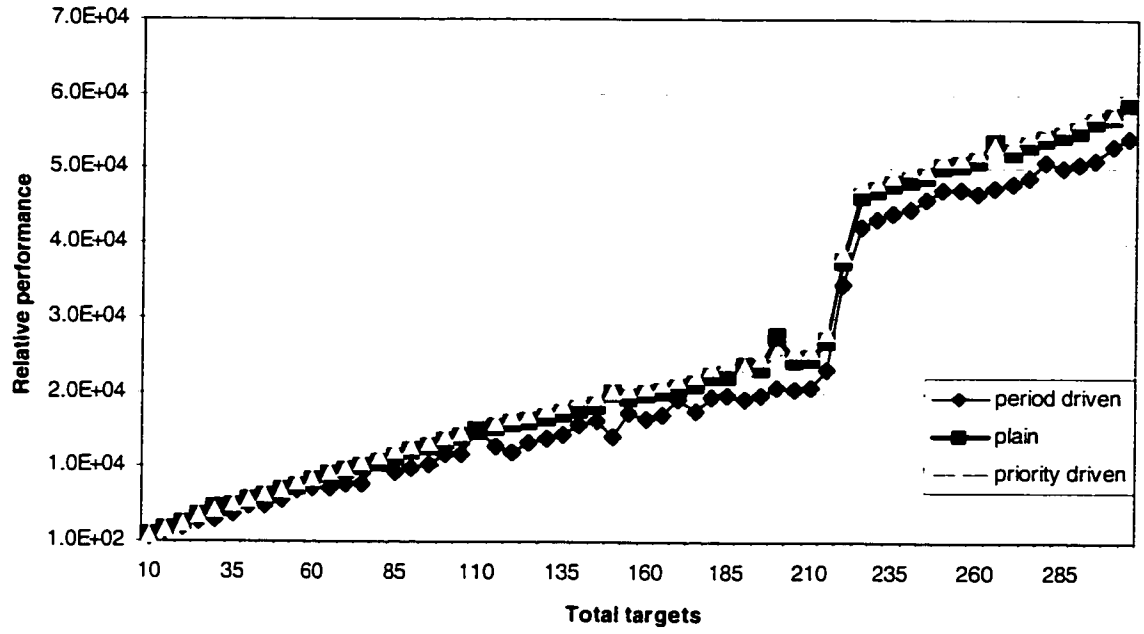

Figure 16. Performance with No load

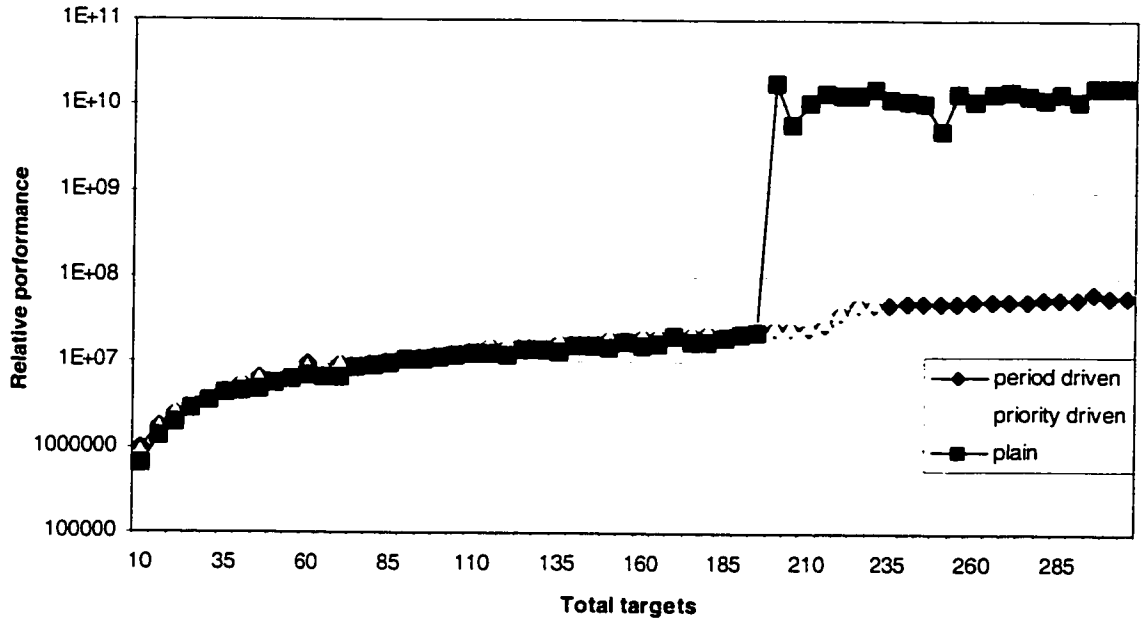

Figure 17. Performance with Light load

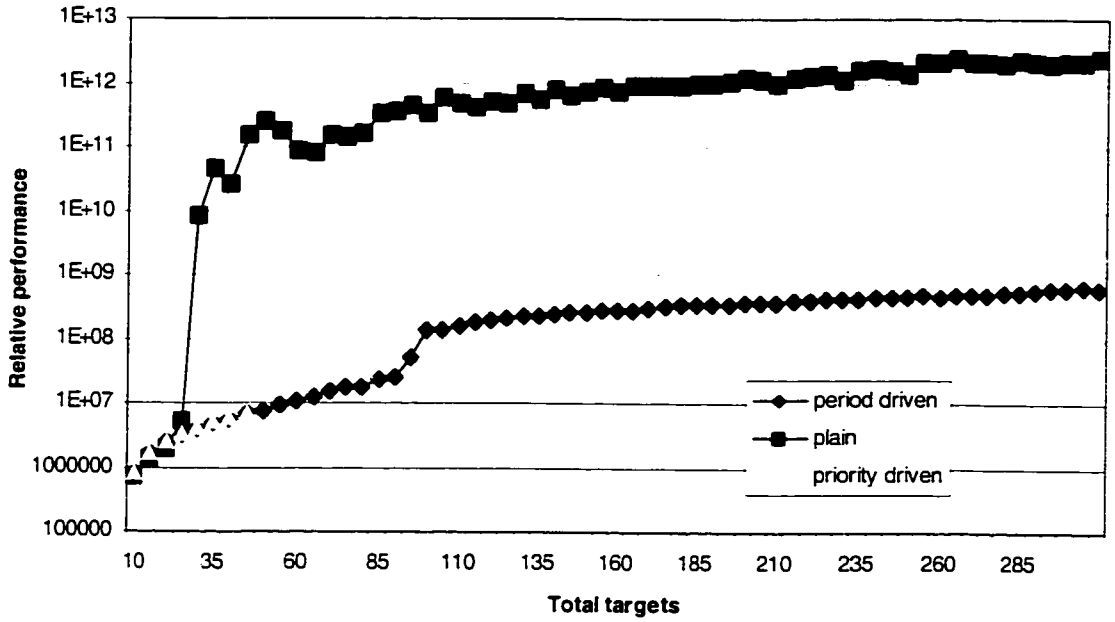

Figure 18. Performance with medium load

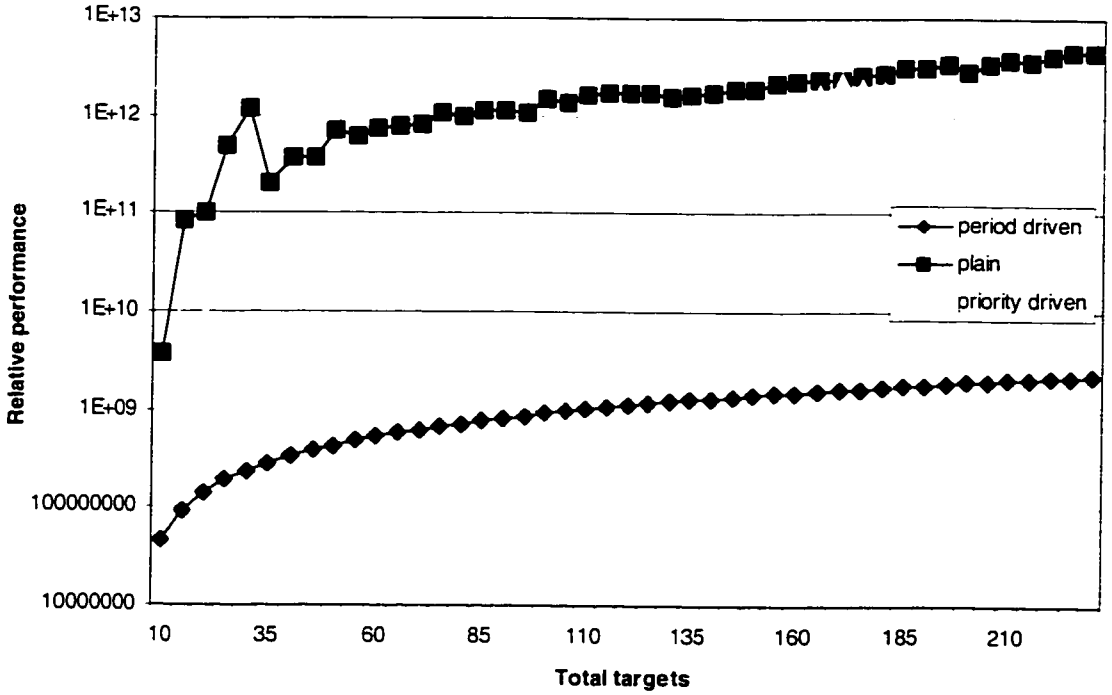

Figure 19. Performance with heavy load

A comparison of Figures 17 and 18 suggests that if the calculation load is small, the priority mechanisms slightly worsen the performance comparatively to the plain system (due to the significance of priority change overhead), but with a heavier calculation load, the priority driven system might already be a better choice than the plain one. The period driven system in Figure 18 loses performance significantly only around 100-target level, but remains to be the most effective among all three systems.

Finally, Figure 19 shows the results with heavy calculation load and as expected shows a rapid performance decrease at a very low number of targets for all the systems. While the priority driven system remains better than the plain, the period driven is significantly better than the other two throughout all the experiments.

Generally, the performance decrease during an increase in number of targets can be explained by the inability of a system to track defined targets quickly and to move a target from the Undefined to the Defined state fast enough due to a big number of existing targets. Also, since the actual period of tracking a target is increasing, more and more targets are lost. Such targets are moved back to the Undefined state, they are getting higher priority than the Defined ones and smaller period of calculation, therefore they consume more time while contributing to big error values.

The plain system does not perform well, because it loses targets arbitrarily, thus leaving a high possibility to lose, or to track with low precision the very important close and fast moving targets. The priority approach would help with a medium to heavy calculation load in comparison to the plain system, but due to the big cost of such implementation (resetting priorities for timers), this system yields to the period driven one, where the special custom timer service is fast and efficient.

In order to easily compare all the results on a single plot, Figure 20 shows the relative performance of all the systems under the different calculation loads

for three different target settings (10, 100, and 150). It is easy to see that in general, at various number-of-target levels and at different calculation loads the performance of the period driven system is much better than the performance of the plain and the priority driven systems whenever the load reaches a point where it causes overload on the system.

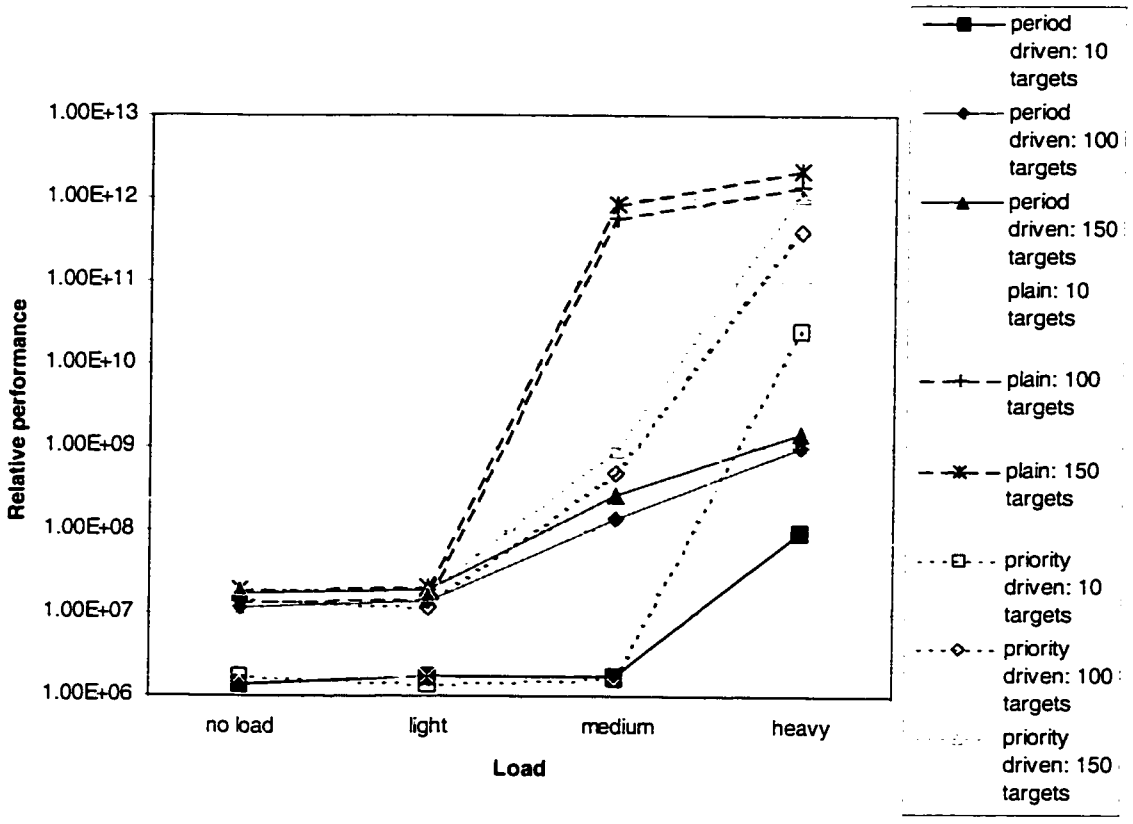

Figure 20. Relative performance vs. load at different max. target levels

5.5 Summary

In this chapter the set-up of all the experiments was presented. We also explained how we varied the load scenarios during the experiments so that we could measure the effectiveness of all three approaches for graceful system degradation under overload conditions. The experimental results proved that for our system the period driven approach was most efficient.

6 Conclusion

In this thesis we applied the ROOM-based CASE tool ObjecTime Developer to design a radar system simulator focusing on the target tracking aspect of the system. Special attention was paid to the performance of the system as the load on the system varied and especially when the system would go into an overload situation.

We found that the ROOM modeling language and its semantics (such as notions of actors, communication ports, state machines) was, in general, highly effective in developing a system of this complexity. It provides all the necessary means for problem abstraction and inter-object communication – this greatly reduced the implementation effort. Furthermore, it is easy to visualize the whole model and then just 'draw' it with the tool to get to an executable state that greatly facilitated development. Some adjustments within the model can be done 'on the fly'. Overall, CASE tools are a big aid to system designers. We are sure that an implementation of our project within a reasonably short time would be almost impossible for one person coding manually only.

However, the actual tool is limited in its other capabilities:

- While it supports the mechanisms for concurrency (threads), communication (messages), and timers, it lacks some elementary mechanisms for manipulating the priorities easily – a feature that most real-time developers would find essential.
- The tool is optimized for mapping the entire design into a single operating system task/thread. However, real-time developers often need to create and manage multiple operating system tasks to benefit from preemptive scheduling of tasks and to deal with blocking behaviours. While the tool provided this capability, the mechanism to place actors on threads was cumbersome and not exposed at the modeling level which made it difficult to change thread assignments easily.

- Many elementary data structures, such as integer, real, etc., are defined as objects, which creates unnecessarily lengthy code. Such an implementation could not be fast enough for real-time systems and not small enough for embedded systems.

The second point of interest of our work – policies for scheduling of tasks in an overloaded environment – showed us interesting results. We have a number of complicated tasks to be performed in a soft real-time system during overload condition. If we do not utilize any special scheduling policy, more important tasks could be postponed in favour of less important ones, which has significant impact on the overall performance. Thus, we were comparing performance of systems with three different scheduling policies: ObjecTime native policy (or 'do-nothing' approach), priority based approach and execution period based approach.

The classical approach would be giving priorities to those tasks, based on some importance function (which should be derived for each particular system). The drawbacks here are the costs of priority changing mechanisms. Thread priority change is already expensive, and many threads would pose very big context switch overhead. In this work we used the 'priority of the message' feature of ObjecTime. Perhaps it would perform better if the tool offered us a mechanism to change priority of timer signal on the spot, but in our case we had to use the full timer reset with a different priority per task approach. This also posed significant overhead. Nevertheless, the priority approach yielded overall better performance than the 'do-nothing' approach, due to the fact that it always favours more critical tasks.

We developed a custom execution period based approach that worked around the limitations of the tool and made use of the application semantics. This required creating a custom timer service that was easy to build and was low in overhead and did not require complicated recalculations or priority changes. The scheme was able to successfully maintain performance even as the system went into high overload situation – thus achieving the goal of graceful degradation.

According to our experimental results, the period of execution approach yields much better performance than the do-nothing and priority approaches.

Our work can also be extended and continued in various ways. New CASE tools will appear and existing ones will have newer versions. They can also undergo somewhat similar case studies, where their usability and effectiveness will be assessed. Radar systems could be case studied deeply according to the specific application and working environment. Special cases of target behaviour as well as special types of targets could be studied. Importance function and period calculation technique can also be studied deeply and from various aspects.

References

[1] Aldarmi, S., Dynamic Value-Density for Scheduling Real-Time Systems, To appear at *The 11th Euromicro Conference on Real-Time Systems* (ECRTS99), York, England, June 9-11, 1999

[2] Biernsos, G., *Optimal Radar Tracking Systems,* John Wiley & Sons, New York, 1990.

[3] Blake, L., *Radar Range-Performance Analysis,* Lexington Booksn, Lexington, 1980.

[4] Bogler, P., *Radar Principles with Applications to Tracking Systems*, John Wiley & Sons, New York, 1990.

[5] Brookner, E. (ed.), *Aspects of Modern Radar*, Artech House, London, 1988.

[6] Burns, A. The Meaning and Role of Value in Scheduling Flexible Real-Time Systems, *Journal of Systems Architecture,* pages 305-325, issue 46, 2000.

[7] Clark, R. An Adaptive, Distributed Airborne Tracking System, *Workshop on Parallel and Distributed Real-Time Systems*, San Juan, PR, April 99.

[8] Hovanewwian, S., *Radar System Design and Analysis*, Artech House, Dedham, 1984.

[9] Krasovec, G., Target Tracking: a Real-Time Object Oriented Design Experiment. In *Proceedings, First International Conference on Complex Computer Systems*, November 1995.

[10] Lee C., Shih C., Sha L. Service Class based Online QoS Management in Surveillance Radar Systems *In IEEE Real-Time Systems Symposium,* December 2001.

[11] Mosse, D., Value-Density Algorithms to Handle Transient Overloads in Scheduling. *European Conference in Real-Time Systems*, (1999).

[12] Rodziewicz P. *Timing and Schedulability Analysis of Real-Time Object-Oriented Models.* Masters Thesis, Concordia University, 1998.

[13] Saksena M., Ptak A., Freedman P., and Rodziewicz P. Schedulability Analysis for Automated Implementations of Real-Time Object-Oriented Models. In *Proceedings, IEEE Real-Time Systems Symposium*, December, 1998.

[14] Saksena M., Rodziewicz P., and Freedman P.. Guidelines for Automated Implementation of Executable Object-Oriented Models for Real-Time Embedded Control Systems. In *Proceedings, IEEE Real-Time Systems Symposium*, December, 1997.

[15] Schleher, C., *Introduction to Electronic Warfare*, Artech House, New York, 1986.

[16] Toomay, J., *Radar Principles for the Non-Specialist*, Van Nostrand Reinhold, New York, 1989.