Discrete Event Simulation Using ADA

Diana ElRabih[1]
[1]*(Research & Development Department/ Monty Holding Company, Lebanon)*

ABSTRACT: *Discrete event simulation has been used to model and evaluate engineering systems. Most engineering processes can be described as a sequence of separate and discrete events. This paper describes the suitability of ADA for discrete event simulation, by showing a case study for developing a discrete simulation environment using ADA and by showing advantages of using ADA. A brief introduction is given to ADA and the reasons why it is considered important as a methodology for discrete event simulation supported in the software environment for ADA. The design of a system or a process often needs to be evaluated for correctness and engineering properties before its implementation. Simulation is a cost-effective mechanism to evaluate system and process design.*

ADA is a modern programming language designed for large, long-lived applications and embedded systems in particular where reliability and efficiency are essential. ADA is a design language as much as a programming language. ADA is designed to be read by ADA programmers and programmers not knowing ADA. In addition to being a programming language, ADA can be used as a communication language for some aspects of the needs and for some aspects of the design. In this paper, discrete event simulation is organized by making use of ADA's structuring concepts

Keywords*-Discrete Event Simulation, ADA, Systems Modelling*

I. INTRODUCTION

Most engineering processes can be described as a sequence of separate and discrete events. Using discrete event simulation modeling, the movement of a train from point A to point B is modeled with two events, namely a departure and an arrival. The actual movement of the train would be modeled as a time delay between the departure and arrival events. These events and movement between them can be smoothly animated. Discrete event simulation focuses on the processes in a system at a medium level of abstraction. Discrete event simulation modeling is widely used in the manufacturing, logistics, and healthcare fields. When the system under analysis can naturally be described as a sequence of operations, discrete event modeling techniques should be used. However, it is not always clear which of the three modeling paradigms is best for a system. For example, if it is easier to describe the behavior of each individual object than trying to create a global workflow, agent based modeling may be the solution. Similarly, if you are interested in aggregate values, and not individual unit interaction, system dynamics may be applied. In classic discrete event tools, the entities are passive and can only have attributes that affect the way they are handled. For example, in a manufacturing model, a crane can be modeled as a resource in a process flowchart, but at the same time have state changes inside, including idle, moving, turning, loading, and so on. Discrete event simulation is a modeling technique for the study of systems whose state may change only at discrete points in time. This type of modeling is applicable to many real-world problems, and a wide variety of languages have been developed for the purpose of modeling these situations. Discrete event simulation has been widely used to model and evaluate computer and engineering systems and has been an on-going area of research and development. The design of a system or a process often needs to be evaluated for correctness and engineering properties before its implementation. Simulation is a cost-effective mechanism to evaluate system and process design. Likewise, to study the behavior of an existing system or process, simulation is often more cost effective than direct system or process measurement. Depending on the underlying system model, the simulation will take the form of solution of a set of equations, as in the case of a continuous system model, or execution of event-based program code, as in the case of a discrete-event system model. In this paper we will consider only simulation of discrete-event systems. ADA is a modern programming language designed for large, long-lived applications and embedded systems in particular where reliability and efficiency are essential. In fact, ADA is a design language as much as a programming language. It is designed to be read by ADA programmers and programmers not knowing ADA. In addition to being a programming language, ADA can be used as a communication language for some aspects of the needs and for some aspects of the design with its embodiment of modern software engineering principles. ADA has rigid requirements for making entities such as subprograms and variables visible globally. This leads to a separation of ADA code into specifications or "specs" and bodies. In general, design begins with a functional description or specification of the object to be designed. The design must then be performed in such a manner as to match the specification. In some areas, particularly software design, the specification process and techniques to ensure consistency between a specification and a design have been highly formalized and automated. ADA is a strongly typed, block-structured programming language with support for concurrent execution, separate compilation, reusable software modules, and extensive compile and run-time verification. ADA defines procedures and functions in the usual way. Tasks are units of concurrent execution. For example, robot programming languages are emerging from their experimental stage and entering an assessment phase. Their main features are illustrated and a parallel with ADA is proposed. The comparison is positive for ADA, in the sense that ADA provides most of the required capabilities. The ability of reasoning on object models and taking decisions will play an increasing role in the future. The use of ADA to program a robot-based manufacturing cell, an example of real-time embedded system, is described in [1]. The computing issues in manufacturing cells are discussed with respect to ADA. Using an experimental manufacturing cell presently under construction as an example, a strategy for robot programming based on ADA is described in [1]. The principal advantages and difficulties in using ADA for programming robot-based manufacturing cells are summarized on the basis of the software issues described in [1].

II. MATERIALS AND METHODS

Simulation technology holds tremendous promise for reducing costs, improving quality, and shortening the time-to-market for manufactured goods. Unfortunately, this technology still remains largely underutilized by industry today. Potential simulation impact areas are closely intertwined with strategic manufacturing. Yet, a number of factors currently inhibit the deployment of simulation technology in industry today. The development of new simulation interface standards could help increase the deployment of simulation technology. Interface standards could improve the accessibility of this technology by helping to reduce the expenses associated with acquisition and deployment, minimize model development time and costs, and provide new types of simulation functionality that are not available today. Discrete-event simulation has been widely used for modelling and evaluating computer systems, computer networks, real-time systems, distributed systems, database management systems, manufacturing systems etc. For example, to evaluate the configuration of a computer system for a banking application, to evaluate a resource management policy in an operating system, to study the behavior of a local area network communication protocol, or to examine strategies for job shop scheduling. Given that several engineering disciplines have found the need for discrete event simulation, several languages and packages have been developed, both for general purpose use, as well as for a suitable set of applications. Advances in software development such as object-oriented design, data structures, and graphical user interfaces have caused advances in simulation techniques and software. Since discrete event simulation is an important field in its own right, a number of languages have been designed specifically for the purpose. In some cases general purpose languages have been enhanced. There is also a common trend to provide application programming interfaces for discrete event simulation in a general purpose high level language. The ADA language was designed to present a general language, unifying, standardized and supporting the precepts of software engineering. ADA is beginning to prove itself of reliability, robustness but has youthful defects. From 1990 to 1995 the revision of the standard leads to ADA95, which corrects small defects, fills a big lack by making the language completely object (ADA is the first object language normalized). ADA95 adds its lot of novelties still unpublished 10 years later. Today ADA does not seem to have the place he deserves especially in first learnings of computer science where we must mix the programming itself with the good practice of programming. ADA is well used (even unavoidable) in avionics and embedded computing (rocket Ariane for example), as well as for traffic control (air, rail) where reliability is crucial. It is also appreciated when the code to develop is consequent (so very difficult to maintain). But the fact remains that currently few small or medium-sized companies admit to using ADA. Modest, productivity gains with ADA are proven and very significant.

III. ADVANTAGES OF ADA

ADA appears more cost-effective compared to other similar languages [3]. ADA, unlike other languages which grew by gradual addition of features, was designed as a coherent programming language for complex software systems. In many instances in other similar languages to ADA such as C language, rules require a non-trivial amount of code development and verification, while the ADA solution is trivial [3]. For instance, achieving object initialization in similar languages requires the use of carefully implemented constructors, while specifying default initialization for ADA records is relatively trivial [3]. Another example is multi- threading with several rules for the use of locks, and condition variables. For ADA, the built-in facilities for direct task communication with protected objects for communication through shared buffers, includes implicit control of locks, and condition variables [3]. The ADA language was developed to satisfy a formal set of requirements. This ensured that from the very beginning the ADA language included the necessary features for its intended applications. The language proposal was published for scientific review before it was fully implemented and used in applications. Many mistakes in the design were corrected before they became entrenched by widespread use. The standard was finalized early in the history of the language, and facilities were established to validate compilers against the standard. Adherence to the standard is especially important for training, software reuse and host/target development and testing. Execution of the statements contained in a task proceeds independently of the rest of an ADA program except at specifically designated synchronization points [3]. Program and data objects may be grouped together in a package. Objects so grouped share identical scope and visibility. In ADA, packages can be used as a passive grouping mechanism. Extensive compile-time checking is performed to ensure that formal and actual parameters do not conflict, and run-time checking is performed to ensure invalid assignments and references are detected. ADA supports separate compilation since it does not force an entire program to be recompiled when a change is made to a single module. Programming in ADA is not, of course, a substitute for the classical elements of software engineering. ADA is simply a better tool. The software engineers design their software by drawing diagrams of the package structure, and then each package becomes a unit of work. Many, if not most, careless mistakes are caught by type checking during compilation, not after the system is delivered. Software integration is effortless, leaving them more time to concentrate on system integration. Though ADA was originally intended for critical military systems, it is now the language of choice for any critical system [3].

IV. CASE STUDY

Simulations are invariably used during the design of large embedded systems. You cannot ‘debug’ the design of a rocket by launching rocket after rocket! It takes months of time and tens of millions of dollars to build each rocket, so extensive simulation is the only way to develop enough confidence in the design to build and launch one with a reasonable chance of success. The case study is a framework for a simulation of a rocket. Needless to say, we will omit all the physical calculations that would require domain-specific knowledge. The choice of a rocket is arbitrary; the framework can be used for any simulation. The method used is discrete event simulation. In this method, events are generated and placed on a queue (Figure 1).

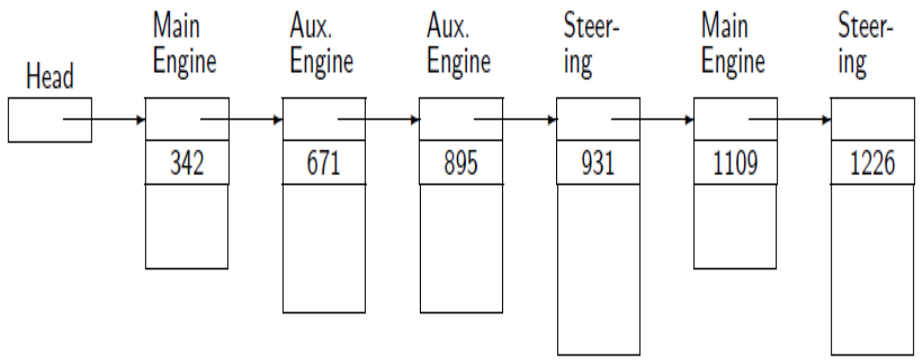

Figure 1. Event queue

Each event is time-stamped with the time at which it is to ‘occur’. The program does not attempt to maintain a physical clock. Instead, the events are ordered by time, and the program simply removes the event whose occurrence is ‘soonest’ in the future, sets the simulated clock to the event’s time and performs the simulation of the event. This is easily done by maintaining a priority queue, where higher priority is given to earlier events. The events themselves would normally be generated by additional tasks in a multitasking implementation. For now, we simplify the program and specify that all events are generated and placed on the queue before the simulation is commenced.

The outline of the simulation is given by the following program fragment:

```
forAll_Events loop
Put(Create_Event, Q);
end loop;
whileQueue_Not_Empty loop
Simulate(Get(Q));
end loop;
```

We have a major problem to solve: how do we represent an event? If all events were identical, a simple record would suffice. But, as shown in Figure 1, different events will have different components associating with them. Some components are, of course, identical in all events: the link and the simulation time. However, the data actually required to conduct the simulation depends on the specific event type. For example, an engine event will need the fuel flow rate, while a steering event will need the deflection angles. The simplest solution to the problem is to include all possible components in a record. This is clearly impractical, because the records will be too large and confusing. Each record will contain only the components required by the specific event type; when accessing the record, you must explicitly select the correct variant using a case statement. This solution can make maintenance difficult, because if you add an event, you must modify every case statement that selects according to the record variant.

A better solution is to use inheritance. We will define a general event type and then specialize the type for each specific event. To use the correct terminology, we will *derive* the specific event types from the *parent* type. The parent type will contain the components that are common to all events, and these will be inherited by the derived types, which will be *extended* with event-specific components.

Tagged Types in ADA

We start by declaring in package Root_Event a record type Event containing a single component Time. Event is an abstract data type, because it is declared to be private with the completion in the private part of the specification below:

```
packageRoot_Eventis
- - Declaration of abstract event at root of event class.
typeEvent is abstract tagged private;
- - Declare (abstract) primitive operations of an Event.
functionCreate return Event is abstract;
procedureSimulate(E: in Event) is abstract;
- - Comparison of events is common to all events in the class.
function"<"(Left, Right: Event'Class) return Boolean;
private
subtypeSimulation_Timeis Integer range 0..10_000;
typeEvent is abstract tagged
record
Time: Simulation_Time; - - Common component of all events
end record;
endRoot_Event;
```

The type is declared to be *abstract*. You cannot declare an object of an abstract type. This is reasonable because a record with just the Time component doesn’t actually represent an event that can be simulated. The type will serve only as the root of a class of types, one type for each event. If you can’t declare an object of the type, you don’t have to have an operation for it. The word tagged indicates that the type can be *extended*. We now extend the abstract type Event for each concrete event type that is needed in the simulation. Package Root_Event.Engine declares three types: Engine_Event derived from Event and two events Main_Engine_Event and Aux_Engine_Event, which are in turn derived from Engine_Event.

```
packageRoot_Event.Engine is
typeEngine_Event is new Event with private;
- - Override primitive operations of Event.
functionCreate return Engine_Event;
procedureSimulate(E: in Engine_Event);
typeMain_Engine_Event is new Engine_Eventwith private;
functionCreate return Main_Engine_Event;
```

```
typeAux_Engine_Event is new Engine_Eventwith private;
functionCreate return Aux_Engine_Event;
procedureSimulate(E: in Aux_Engine_Event);
private
typeEngine_Event is new Event with
record
Fuel, Oxygen: Natural;
endrecord;

typeMain_Engine_Event is new Engine_Eventwith
null record;
typeAux_Engine_ID is (Left, Right);
typeAux_Engine_Event is new Engine_Eventwith
record
Side: Aux_Engine_ID;
endrecord;
endRoot_Event.Engine;
```

The types derived from Event can be displayed in a tree (Figure 2). The components of a derived type consist of the components *inherited* from the parent type, as well as any additional components added in the extension. Since there is no provision for removing components upon derivation, we can be sure that *every* component of a parent type, for example Oxygen in Engine_Event, is also contained in each descendant of the parent: Main_Engine_Event and Aux_Engine_Event.

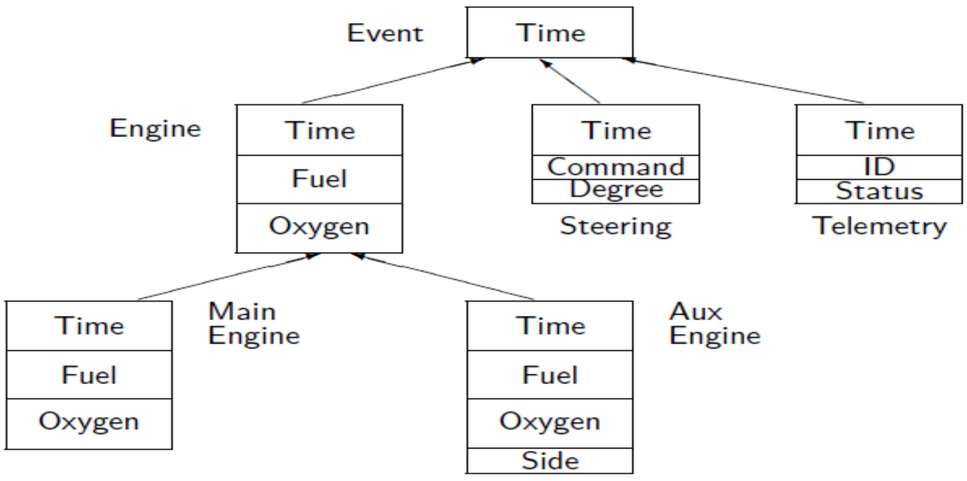

Figure 2: Types derived from Event

Note that the full view of the private extension can be indirectly derived from the ancestor in the following example, the partial view of Main_Engine_Event is derived from Event, while the full view is derived directly from Engine_Event and only indirectly from Event.

```
packageRoot_Event.Engineis
typeEngine_Event is new Event with private;
typeMain_Engine_Event is new Event with private;
private
typeEngine_Event is new Event with . . .
typeMain_Engine_Event is new Engine_Eventwith. . . ;
endRoot_Event.Engine;
```

Moreover, it does emphasize that in ADA, decisions relating to encapsulation (packages) are *independent* of decisions relating to derivation classes of types. This point is worth emphasizing because many languages for object oriented programming identify a type with its encapsulation in a ‘class’. The limitation that a client can only see the visible part of a package is too inflexible for some applications. Consider the declaration of Root_Event.Event: on one hand we wish to restrict the accessibility of the implementation of the type (the component Time), while on the other hand, derived types should be able to access this component because computations within Simulate may be time-dependent. The solution is to use *child packages* to form a *subsystem* of packages. Packages within the subsystem share abstractions by granting child packages visibility of the private parts of their specifications. Child packages are denoted syntactically by concatenating the child name to the parent’s name using dotted notation. The hierarchy of descendants may be carried to any depth.

Moreover, ADA defines two types of child packages: *public* and *private*. The rule in ADA fragments holds only for public children; the visible part of a *private* child *is* allowed access to the private part of a parent; however, to prevent unwanted exportation, a client of a private child must be within the family that already has access to the private part. Consider the random number generator package Root_Event.Random_Time. The package is declared as a generic instantiation of a package from the standard libraries, but in the context of our simulation program, an equivalent specification is as follows:

```
private package Root_Event.Random_Timeis
typeGenerator is limited private;
functionRandom (Gen: Generator) return Simulation_Time;
procedureReset (Gen: in Generator);
private
. . .
endRoot_Event.Random_Time;
```

The function Random in the visible part of the specification returns a value of type Simulation Time that is declared in the private part of its parent, in effect exporting the type. This is not normally acceptable, because Simulation Time was intentionally made private to prevent its exportation from the simulation subsystem.

V. CONCLUSIONS

In this paper, we consider a case study of discrete event simulation using ADA showing advantages of ADA. The design of a system or a process often needs to be evaluated for correctness and engineering properties before its implementation. Simulation is a cost-effective mechanism to evaluate system and process design. Discrete-event simulation has been widely used for modelling and evaluating computer systems, computer networks, real-time systems, distributed systems, database management systems, manufacturing system ADA is a modern programming language designed for large, long-lived applications and embedded systems where reliability and efficiency are essential. This paper show a case study for developing a discrete simulation environment in ADA. The introduction of simulation facilities in ADA not only concerns the classical aspect of model building, but allows a new class of problems to be tackled, that is the testing of correctness of programs intended for real-time applications.

REFERENCES

[1] Ben-Ari, M. ADA for Software Engineers. *Weizmann Institute of Science*. 2005.

[2] A. Wearing, Software Engineering, ADA and metrics, *LNCS Volume 3,* 2005.

[3] S.F. Zeigler, Comparing Development Costs of C and ADA, *Rational Software Corporation*, 1995.

[4] G. Booch and D. Bryan, Software Engineering with ADA, *3rd Edition, Addison-Wesley Professional*, 1993.

[5] V.A.Downes, R. Tellaeche Bosch, Discrete Event Simulation with ADA, *UKSC Conference on* Computer Simulation, Science Direct, 1984.