AN ACTIVITY ORIENTED VISUAL MODELLING LANGUAGE
WITH AUTOMATIC TRANSLATION TO DIFFERENT PARADIGMS

LUÍS M. SILVA DIAS
<lsd@dps.uminho.pt>

A. J. M. GUIMARÃES RODRIGUES
<agr@dps.uminho.pt>
GUILHERME A. B. PEREIRA
<gui@dps.uminho.pt>

Department of Production and Systems
University of Minho
Braga, 4710-057, Portugal
<http://www.dps.uminho.pt>
Phone: +351253604740/+351936271733 Fax: +351253604741

ABSTRACT

The traditional approach for discrete event simulation modelling includes visual support diagrams for modeller-client communication purposes (model interpretation and validation) and also to act as the basis for simulation language program construction.

Although modern simulation packages use powerful graphical interfaces for programming and animation purposes, these packages still require enormous simulation expertise to construct a simulation program. This work suggests the use of the Activity Cycle Diagrams-ACD (activity based philosophy) concepts for modeller-client communication, but also to act as an automatic generator of simulation programs under different paradigms - event scheduling (Basic Simulation Facility – Simulation Library) and Process Flow (ARENA – Simulation Environment) philosophies, thus eliminating any programming effort and expertise.

1 INTRODUCTION

The use of visual support diagrams to help the programming step of a simulation project is very common. Even when generic programming languages were replaced by specific purpose simulation languages the use of paper diagrams remained as a previous step to programming. [34] [25]. These diagrams were conveniently abstracted serving as support to the communication between the simulation client and the modeller (simulation expert), but also helping the construction of the corresponding computational programs [6] [29].

Modern simulation languages introduced new powerful graphical interfaces, but these interfaces are clearly programmer oriented, raising the difficulty in communicating with the client and still requiring enormous simulation expertise to use them [11].

In this paper, we still suggest the use of a (simple) graphical support as a representation of what the client formally needs, but these diagrams will also act as the source to the automatic generation of simulation programs.

The visual language chosen is the Activities Cycle Diagrams (ACD), for its simplicity and efficiency in representing real operating systems. [10] [25]

This mechanism also implied the construction of translation grammars. These grammars were written according to a modular specification of visual languages, based on attribute grammars [15] in MASOViLa (Modular Attribute-based Specification Of Visual Languages) [27] [9] [36]. Our translation engine uses a pattern matching rewriting mechanism.

Using Activity philosophy for modelling, then generating simulation programs based on event philosophy and in process philosophy, three major simulation approaches [4] were explored and linked.

2 LAYOUT AND ANIMATION ORIENTED SIMULATION ENVIRONMENTS

As already referred, appropriate diagrams were in use for many years to support the communication among people interested in a particular simulation. The simulation expert would then translate these models into a simulation language or even a general purpose programming language. As far as graphical support became available, an
enormous variety of simulation environments emerged.[16]. **Graphical facilities** were then used to essentially represent a **system layout for animation purposes**.

Animation is recognized as an important aspect of simulation. However when the modelling process is **focused on animation**, several disadvantages may arise:

1. The **model may be overwhelmed** by many modules and accessory configurations.
2. Such a model, with increased complexity, will be **difficult for the client** to understand.
3. The analysis of a static model (in the first stages) based on the layout will not add much over its photograph or scheme. Thus the **semantic validation** will be left for the **animation phase**.

### 3 PROJECT MAIN STEPS

The main steps of this research work are summarized below:

1. The **choice of an easy** to use and widely spread visual language: **Activity Cycle Diagrams (ACD)**.
2. **Formalization of ACDs** (keeping it simple), allocating to each graphical object the information required by the model.
3. Specification of a **file format** to represent the referred graphical objects (XML).
4. Implementation of a **graphical editor** to draw the models.
5. Implementation of a compiler’s **compiler** for visual/graphical 2D languages.
7. Implementation of **two compilers** - the first compiler generates the simulation program code in **JavaBSF**, and the other one generates the **Arena** program (using both Arena modules and new developed modules in VBA code).

#### 3.1 Programming tools

The main programming language used was **Java** [21] [5]. Several sets of classes were implemented (corresponding to about nine thousand lines of code). Visual Basic for Applications (VBA) was also used, both in ARENA and in Microsoft VISIO. A template was built in ARENA (Professional Edition) [18] with activities and queues to implement an activity-based executive on a process-oriented environment.

### 3.2 About the use of Graphics in Visual Modelling and Programming

The idea behind this work is to enhance the **utilization of graphical facilities in modelling**, making it a great contribution to **automatic generation of simulation programs**, keeping it simple and portable.

Graphical facilities are more helpful when they **support model semantic** than when it is based on system layout and animation.

Furthermore the utilization of a **simple** axiomatic set may be **accessible** to more potential users and clients.

The use of an **activity-based philosophy** seems to be semantically richer and more simulation oriented than process flow or event scheduling.

The strategy was to create a completely **open system**, since the graphical editor creates a text file in XML (eXtended Markup Language). This is compatible with any graphical editor using the established syntax. The compiler uses an XML file and a grammar to build a program in an object simulation language.

### 3.3 Translation

Our first **ACDs compiler** generates a Java program according to the **event scheduling** philosophy. Although this is computationally highly efficient it is harder to program. In other words we could say that it is more computer friendly than programmer friendly. The complex compiler developed allows an easier translation by the computer.

By developing a template with new blocks for **Arena (process oriented)**, we were able to simulate ACDs using Arena objects built according to an activity based philosophy.

Using these two compilers/translations, we explored deeply on three major modelling philosophies [25] [29], making automatic translation mechanisms between them.
4 PROJECT MAIN TASKS

The above diagram illustrates the interdependencies between the main tools and contributions of our project:

- **ACD.xml** «in the centre of diagram» represents the chosen format to physically support the models: a text file in XML. (See also Figure 13 and Figure 14)

- **GRAPHEDITOR** «upper left zone of the diagram» is the graphical editor that was specifically built for this purpose (see also Figure 9). Microsoft Visio was also customised to deal with ACD XML files – VISIO-ACD.

- **ANIM VISIO** - «upper right zone» simulates and animates ACDs inside MS VISIO. The ANIM-Java tool will animate ACDs in the GRAPHERITOR (is not yet completed, as signalled in the diagram by white background).

- **SMAIS-GAP** is the major tool of this project «lower left section» allowing the translation of an ACD (activity-based) into a program (event-based) (see also Figure 15, Figure 16 and Figure 17). It uses the ACD Grammar (to BSF) (see also Figure 10 and Figure 12).

- **SMAIS-GAP** using the ACD Grammar (to Arena) «lower right section» refers to the translation to Arena

input file. ARENA-CREATOR, using that input file, builds a model with blocks from ARENA-ACD template (see also Figure 18, Figure 19 and Figure 21).

5 ACTIVITY CYCLE DIAGRAMS (ACD)

ACDs were widely used mainly with older languages (e.g. ECSL [6], HOCUS) to schematically specify the system’s behaviour, through each assumed entity cycle diagram. These entity cycles explicitly refer the active states into activities and passive states into queues. This graphical language just requires the use of three types of graphical objects: Rectangles (activities), Circles (queues) and Arrows (links).

For an activity to start, it is necessary that entities exist in the preceding queues in the required number and with the adequate attributes. When these conditions hold, it is possible to start the activity. When the activity ends, the entities involved are moved to consequent queues.

The complete model consists of the Activity Cycle Diagrams of all the classes of entities, together. Interactions between entities take place at activities. Figure 2 represents the basic activity concept, with one activity in the middle that starts when each precedent queue has one required entity. When the activity ends, the entities are moved to the consequent queues.

Figure 1: Project Map Diagram

Figure 2: Activity concept

The simplicity inherent to the activity concept and to the ACD facilitates its easy understanding for validation and teaching purposes. It has been advocated that the ACD is useful for research discrete event simulation studies.
5.1 Example: the Bartender Problem

In this illustrative example we have a barman that serves customers in a bartender [6] [29].

Entities of class CUSTOMER, are initially OUTSIDE. They ARRIVE and then WAIT for activity POUR. When served they are READY to DRINK. After that, if they NEED to drink more they go to queue WAIT, otherwise they leave to OUTSIDE. The ACD of CUSTOMER is described below (Figure 3).

Entities of class BARMAN, are waiting in the queue IDLE and they can either participate in the activity POUR or WASH:

Entities of class GLASS are initially in the queue DIRTY. After being WASHed (in batches of size 3), they wait in the queue CLEAN. When there are 1 customer waiting, 1 barman idle and 1 glass clean, then POUR activity begins. Once FULL, the glass goes to activity DRINK. The ACD of GLASS is in Figure 5.

This problem instance parameters are:

Activities duration:
- ARRIVE: Poisson distribution, average=20.
- POUR: Poisson, average=6 \[\text{poisn}(6)\].
- WASH: Fixed = 5.
- DRINK: 5 + Uniform distribution between 0 and 6 \[5+\text{randnum}(0,6)\].

Entities initial allocation:
- CUSTOMER: 20 in queue OUTSIDE.
- BARMAN: 1 in queue IDLE.
- GLASS: 12 in queue DIRTY.

Entities class setup:
- CUSTOMER: have one attribute: NEED.

Attributes:
- ARRIVE: Customer attribute NEED is initialized with \(\text{(int)}\text{randnum}(0,3)+1 \rightarrow \{1,2,3,4\}\)
- DRINK: Customer attribute NEED is decremented. After this activity, customer attribute NEED is evaluated to decide customers destination.

Simulation setup:
- DURATION: 1000 time units.
- WARM_UP: 120 time units.
- SEED: 123543.

The following ACD (Figure 8) include all system information:
DOOR is an auxiliary entity used to control the Customer’s arrives (one at a time). It is well described in [29].

The following image (Figure 9) shows a screenshot of the GRAPHEditor, editing the bartender problem.

Figure 9: GRAPHEditor Screenshot

5.2 ACD Language Formalization

Some authors suggest modifications in ACD language, (e.g. [8]), but in our opinion that compromises the ACD simplicity.

A significant contribution of our work consisted on the development of the ACD language formalization embedding in the diagrams all the information required for the simulation.

1- We added attributes to the visual objects:
   - ‘duration’ and ‘priority’ in activities. (‘priority’ is a value setting the activity priority over other activities. Higher value means higher priority).
   - Entities attributes changing in activities. (EX: In the DRINK activity of the example model, the customer ‘need’ attribute is decremented by one. (activity attribute ‘entity_customer’ = “need=need-1”).
   - ‘initial_entities’ in queues.

2- In order to include global information on the simulation a rectangle alone (called simulation setup) is used with the following attributes:
   - ‘model_name’ – Model_Name_String
   - ‘duration’ - Simulation time
   - ‘warm-up’ - initialization time.

3-For global attributes and features of each entity, one ellipse alone is created with:
   - ‘entity_name’ – name of the entity
   - ‘total_entities’ – number of total entities of this kind that will exist in the simulation.
   - ‘[attribute]’* - used zero or more times to declare entity attributes. (EX: customers have one attribute: ‘need’).
   - ‘[sort]’ – expression establishing the queue sorting rule. Value is defaulted to 1. If defined, the queue will not be FIFO, this attribute must have an expression, elements are sorted in ascending order based on the evaluated expression over each arriving entity. May be used to create virtually, multi-queues.

4-Arrows

- Arrow ending on activities:
  ‘label’ have the number of need entities (entering throw this arrow in the activity) to start the activity (this implements batches). Default value is 1 (if omitted). (EX: 3 is the batch number of ‘glasses’ to start ‘WASH’).
- Arrow leaving from activities:
  ‘label’ have conditions to decide which destination queue will be chosen (usually based on attributes values). EX: ‘need’ attribute is used to decide if the ‘customer’ go OUTSIDE or WAITING after drinking.

6 THE GRAMMAR AND TRANSLATION

To create the AIMS compiler, we wrote a set of rules in Visual MASOViLa notation [9]. Each rule synthesizes one new symbol. We developed rules for queue (3), for activity (9), for entity (3), for input_link (1), for output_link (1), for simulation (5), and also for syntactical and semantic error detection (10) – see Figure 11.

We include below one expression example, with textual explanation, visual representation and the generated Java code (portion of the compiler code).

The next graph (Figure 10) is the rule that transforms an arrow in an output_link, when it is connected from an activity to a queue. The new output_link symbol, receives all attributes from arrow symbol. Furthermore it synthesizes the Link’s attribute origin from the activity’s attribute name, the attribute destination from the queue’s attribute name and also the condition attribute from arrow label.

Figure 10: Output_link rule written in Visual MASOViLa

Java corresponding code:

```java
public static Symbol output_link(Vector args) { Symbol activity = (Symbol) args.get(0); Symbol arrow = (Symbol) args.get(1); Symbol queue = (Symbol) args.get(2); if(((Arrow)arrow.get("container")).leavesFrom((Container)(activity.get("container"))) && ((Arrow)arrow.get("container")).arrivesTo((Container)(queue.get("container")))) { Symbol res = new Symbol(arrow); res.put("origin", activity.get("name")); res.put("destination", queue.get("name")); res.put("condition", queue.get("label")); res.put("entity_name", queue.get("entity_name")); return res; } else return null; } //& output_link
```

Figure 11: - Java code for Output_link rule.
The Output_link rule is just one example that can be found in the central region of Figure 12: The translation is archived by an engine (the SMAIS-GAP) that successively searches patterns in the Diagram under analysis to try to apply each grammar rule (in a specific order). Applying a rule, means rewriting or creating one symbol in the Diagram (some rules may also remove symbols from the diagram).

To the Output_link rule, the engine must know that this rule needs: one activity, one arrow and one queue. The engine then picks all combinations of symbols from the diagram under analysis, submitting different sequences with (activity, arrow, queue), to the rule. The rule will then return \textit{null} if the symbols are inappropriate, or return a new symbol to the diagram. Attributes of each new symbol are “richer” than previous (collecting pieces of code). For this grammar, after several rule modifications, the generated program is represented within an attribute of a synthesized symbol (simulation).

7 XML FILE INTERCHANGE FORMAT

The objective was to give birth to a proven visual language (an example of a complete program written in this language can be found in Figure 8). This program is not dependent on the translator/compiler used. We defined an XML format (eXtended Markup Language) making possible to store it with all the attributes and to transfer it to any application in a text file (since to keep it in bitmap format (as a photo) would not obviously be appropriated).

In Figure 13 we can find the DTD corresponding to the defined XML format. In Figure 14 we can see a portion of one XML file corresponding to the barman problem. Each file is a book: a collection of sheets. Each sheet contains one ACD diagram. (Obviously, books can have only one sheet).

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Book SYSTEM "ACD.dtd">
<BOOK>
...
</Book>
```

Figure 13: DTD file (XML specification)
Running SMAIS-GAP, using the "ACD GRAMMAR (to JAVA-BSF – AUTOMATIC PROGRAM GENERATION"

Generating the program from the ACD file 'ACD.xml' and including an event-driven executive that uses a java simulation library – BSF (Basic Simulation Facility).

In this Example, SMAIS-GAP read the 36 graphical objects from the file ‘ACD.xml’ and generated a fully commented program with 335 lines in about 10 seconds.

An extract of the automatically generated program from the Bartender ACD, (including all comments) can be found below in Figure 16.

```java
public void begin_of_WASH(){
    //@@@@@@@@@@    Activities BEGIN   @@@@@@@@@@@
    System.out.println(Bsf.report(Clock))
}
```

Figure 14: Part of an XML file (representing one ACD object.

Figure 15: Screenshot of Script running, calling SMAISGAP

Running SMAIS-GAP, using the "ACD GRAMMAR (to BSF)" and giving the XML file as input program, we get the generated "bsfProgram.java", see Figure 15.
Figure 16: Extract of Java generated simulation program

```java
public void end_of_WASH()
{
    r = Bsf.remove(WASH_BARMAN, Clock);  //returns r.time, r.at1, r.at2
    Bsf.insert(IDLE, Clock, Clock, r.at2);
    for(int i=3; i>0; i--)
    {
        r = Bsf.remove(WASH_GLASS, Clock);  //returns r.time, r.at1, r.at2
        Bsf.insert(CLEAN, Clock, Clock, r.at2);
    }
    //Attempts to start subsequent activities that may be viabilized
    begin_of_POUR();  //priority=1
    begin_of_WASH();
    } //& end_of_WASH
```

Figure 17: Screenshot of program execution and final statistical report

This task (automatic translation) is quite complex since it implies different abstraction levels, from an high level (activity based approach) to a much lower level (event-driven approach) using a generic programming language.

9 ARENA MODEL CREATOR

The hard part of this task was to create an activity-based executive over a process-oriented environment. The executive developed shows to be more efficient than a three-phase approach since we used a message passing mechanism that only tries to start an activity when entities have arrived to one of its predecessor queues. Given that we generate a model into a high-level simulation environment it becomes possible to overcome limitation of the activity-based approach since the model may be completed in ARENA.

Figure 18 and Figure 19 illustrate the model as automatically created in Arena for the Bartender Problem and a

Screenshot of animation phase. Figure 20 have details of two activities with predefined statistics. Figure 21 have a screenshot with part of the logic template (code) of the activity block.
CONCLUSION

The work presented in this paper could constitute a major step towards the generalisation of the use of simulation. In fact, we suggest the use of a simple interface (Activities Cycle Diagrams) to model a real situation. Then we present a tool capable of generating a simulation program. Based on event scheduling simulation modelling philosophy, our tool automatically generates a program to use Basic Simulation Facility routines. Based on process flow simulation modelling philosophy, our tool automatically generates an ARENA program. Furthermore the mentioned automatic generation of simulation programs does not require expertise in simulation.

REFERENCES AND BIBLIOGRAPHY


**AUTHOR BIOGRAPHIES**

**LUÍS MIGUEL SILVA DIAS** is a Lecturer at the Department of Production and Systems Engineering at the University of Minho. He has earned a degree in Informatics and Systems Engineering, and a Master in Informatics from the University of Minho. His interests are in modelling & simulation, operational research and visual languages.

<lsd@dps.uminho.pt>
<http://www.dps.uminho.pt>

**ANTÓNIO JOSÉ M. GUIMARÃES RODRIGUES** is a full Professor in the Department of Production and Systems Engineering at the University of Minho. He is, since 2002, the Rector of the University. He has earned a Master’s Degree and Ph.D. from the University of Birmingham in Production Engineering. Prof. Guimarães Rodrigues interests are in simulation, operational research and mathematical programming. He is Associate Editor of Investigação Operacional, the Portuguese Journal of OR.

<agr@reitoria.uminho.pt>

**GUILHERME AUGUSTO BORGES PEREIRA** is a Professor in the Department of Production and Systems Engineering at the University of Minho. He is the director of the industrial engineering and management licenciatura (first degree). He has earned a Master’s Degree in Operational Research, and Ph.D. in Manufacturing and Mechanical Engineering from the University of Birmingham. His interests are in simulation, operational research.

<gui@dps.uminho.pt>

University of Minho was founded in 1973, is located in Braga and Guimarães, in the north of Portugal. Has more than 15000, and around 500 Ph.D. Teachers.

<www.uminho.pt>