Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

THE EFFICIENT IMPLEMENTATION OF WAIT STATMENTS

Ingolf Ståhl

Stockholm School of Economics
Box 6501
S-113 83 Stockholm, SWEDEN

ABSTRACT

This paper gives an overview of some of the factors that determine the efficiency of differently implemented wait statements. The purpose is to give some guidance to simulation modelers as to what system to choose and, for the chosen system, what wait constructs to use in order to make program execution efficient. It is also aimed at informing constructors of simulation software about what issues are important when implementing these statements. In particular, we present some brand new features that can speed up the execution of wait statements, in particular in some versions of GPSS.

1 INTRODUCTION

One of the most fundamental concepts in Discrete Event Simulation is the **wait concept**. This deals with an entity being delayed until some defined condition or set of conditions becomes true. We can e.g. think of a ship that will wait before entering a harbor until there is no storm, and there is a free berth as well as a free tug. The wait command, which we call the wait word, by which an entity, like the ship, is kept waiting, has many different names, like GATE and TEST in GPSS/H, WAITIF in micro-GPSS, WAITUNTIL in DEMOS, WAIT UNTIL in SLX and SCAN or WAIT in SIMAN, just to mention a few examples. The wait word is usually followed by a logical expression that can be either true or false. If due to some event, this condition changes (from TRUE to FALSE or from FALSE to TRUE), one or several entities that have been waiting can at this clock time become the active entity and move forward in the model system.

There is no other statement in simulation systems that can cause such a difference in execution efficiency. We shall below give an example when one use of a wait statement in a system can cause a twenty times longer run time than a better use of the statement, even though we use the same simulation system. It is hence important to know if it is possible within a certain system to use an efficient wait statement rather than an inefficient one. If it is not possible to use a reasonably efficient wait statement, when trying to simulate a specific system, the best thing might be to migrate to another system with more efficiently implemented wait statements. For a constructor of a simulation system or a programmer writing a simulation program in a General Purpose Language it is also of great importance to implement efficiently working wait statements.

It should be stressed that the term efficiency should not be limited to only execution speed. There is also efficiency in terms of time required to program the model, in particular connected to debugging due to errors caused by an improper wait statement construct. Even if the focus is on efficient execution, we shall also touch upon this other aspect of efficiency.

We want to look at the idea of execution efficiency from a **theoretical** point of view and disregard the compilation/interpretation choice and the programming skill of the implementers of a specific system. Obviously, a compiled system, with a compiler written by skillful programmers, using an optimizing compiler, will, *ceteris paribus*, execute faster than an interpretative system, written by a less skillful programmer, using a less expensive programming system. This difference might make a compiling system with a theoretically poor wait statement out-perform an interpretative system with a theoretically superior wait statement. Our hypothetical test of performance should hence assume that the systems are implemented in the same type of system as regards the compiling/interpretation choice, using the same type of software and by the same type of programmers. Such a hypothetical test will, of course, never take place. Hence, our comparisons as regards efficiency must deal with questions of e.g. how many times a search of a certain list will take place and how many values will have to be investigated each time.

This article is in line with a series of articles, started by those of Schriber and Brunner (e.g. 1994 and 1997) and followed, *inter alia,* by one by myself (Ståhl, 1996). The main idea behind these articles is that it is important to know more about how discrete event software works in order to be a good user of these systems. While the other articles mainly focused on how knowledge about the software is important for avoiding logically faulty programming and dealt with several different types of statements, the present article is mainly concerned with the efficiency of one particular type of statement.

The issue of efficient implementation has received renewed attention during the last couple of years, due to the emergence of the SLX simulation language (Schulze & Henriksen 1998). A prominent aspect of SLX is the very efficient implementation of the wait statement. There are hence reasons to look more closely at the main principles of how SLX handles this wait statement, as well as to do some evaluation of it, by comparing it to other ways of implementing the wait statement.

It should be stressed that this overview of issues regarding the wait statement is in no way exhaustive. It can only point at some of the main alternatives among the many combinations possible. It is also limited to those systems for which one in the literature can find a reasonably clear discussion about how the wait conditions are implemented. It also reflects the systems about which I have some knowledge and for which I have access to at least some version of the software.

2 GENERAL BACKGROUND

To facilitate the reading of the paper, we first look at the five states in which an entity can be according to Brunner and Schriber. 1. The **one** single currently moving entity is in the **Active State**. 2. The entities that are ready to enter the Active State at the current value of the simulation clock are in the **Ready State**. 3. The entities that will enter the Ready State at a **known** higher value of the simulation clock are in the **Time-Delayed State**. The entities that will enter the Ready State at an **unknown** value of the simulation clock are in either the Condition-Delayed State or the Dormant State. 4. The **Condition-Delayed State** is the state from which the entities are transferred **automatically**, implying that the modeler does not have to supply specific software logic for getting this done. It will be done by software logic that is supplied for doing primarily other things. 5. From the **Dormant State** the entities are transferred to the Ready State on the basis of **modeler supplied** logic, which has no other purpose than doing this.

The entities waiting in the Time-Delayed State in all systems discussed here wait on a **single Future Events List** and all entities in the Ready State reside on a **single Current Events List (CEL)**. The entities waiting in the Condition-Delayed State wait on different types of lists in the systems, usually in one or several **delay lists**. The waiting in the Dormant State is also taking place on different types of lists, often on one or several **independent user managed lists**. When we below discuss wait conditions, the focus is on delay lists, but we shall also occasionally touch on user managed lists.

The efficiency of the specific delay list approach chosen is in turn connected to the question of whether we have what Brunner and Schriber call related or polled waiting. Both procedures refer to **automatic** removal of entities from the delay list. The **related waiting** resolution implies that the checking of a delay condition is connected to a specific event. The **polled waiting** resolution implies that entities are not necessarily immediately removed from the delay list as a certain event occurs, but will in another process, but at the same simulation clock time, be removed, possibly in the same process for several different waiting conditions. For the case of independent user managed lists, the removal of entities is done by modeler supplied signals that are sent out by specific events.

In order to be better able to discuss the issues of what types of lists and what type of removal procedure to choose in connection with waiting, I also want to introduce some new distinctions. It appears in this context to make sense to distinguish between **simple waiting**, where the entities wait for one **single** event to happen, and **complex waiting**, where the entities wait for more than one single event to happen.

As regards simple waiting, it is suitable to distinguish between simple waiting dealing with common events, usually concerning a resource or a switch, and waiting concerning some other not so common event. Since these common events are frequent, it is worthwhile, from an efficiency point of view, to spend more efforts to have this type of waiting handled in an efficient manner. We hence distinguish between **common simple waiting** and **special simple waiting**. Common simple waiting could be waiting for a machine to get idle, while special simple waiting could be waiting for the number of a certain type of entities to reach a specific value.

As regards common single waiting, we can in turn distinguish between implicit and explicit waiting. In the **implicit waiting** case, the program contains no specific operation starting the waiting, while in the explicit waiting case there is always such an operation. An example of implicit common simple waiting is the waiting in front of a SEIZE block in GPSS and in SIMAN. There is no need to have a QUEUE block in front of the SEIZE block. In both systems entities unable to get into the SEIZE block to use the resource of this block will be put on a delay list, in the case of SIMAN an internal queue. **Explicit waiting** can for GPSS/H be exemplified by a GATE block without a C operand and for SIMAN by the inclusion of a QUEUE block in front of the SEIZE block or by a SCAN block. Thus waiting for a lathe in GPSS can be done just in front of a SEIZE LATHE block, implying implicit waiting, and before some other block, e.g. GATE NU LATHE (in GPSS/H) or WAITIF LATHE=U (in micro-GPSS), implying explicit waiting.

It should finally be noted that in the systems discussed here, special simple waiting and complex waiting **always** require explicit waiting statements.

3 DIFFERENT ISSUES IN DEALING WITH WAITING

There are many different choices to be made as regards waiting when constructing a system or a model of discrete event simulation: 1. the number of delay lists, 2. types of delay lists, 3. when the testing for exit from a delay list shall take place, 4. how many entities shall then be removed from the list at a time, 5. whether or not all waiting shall be explicit and 6. if waiting shall be in terms of WAIT UNTIL or WAIT WHILE.

3.1 Number of delay lists

We can here distinguish between the following three main cases: **A**. There is only one single list for all entities waiting at wait statements. **B**. There is one specific list for each waiting condition, i.e. there are as many delay lists as there are delay conditions. **C**. There is a mixture of these conditions, so that entities from some wait statements are on one joint list, but other wait statements each have their own list.

If there is one single list for all waiting entities and the entities do not all wait for the same condition, a considerable search activity is required. If a certain condition has just changed, e.g. a tug is now free to serve ships, we have to search the single delay list for ships waiting for a tug, among many other entities, held up due to other conditions. Regardless of the choice made in sections 3.3 and 3.4 below, the search procedure can in case the first entity is located far from the starting point of search, usually at the front of the list, be considerable. Examples of systems where all (except user controlled) waiting appears to take place on one single list are GPSS/H, GPSSS (Vaucher, 1977) and many simulation packages, written in a GPL, like Pascal_Sim, using the Tocher three-phase approach (Davis & O'Keefe 1989).

If each waiting condition has its own list, all entities on the list wait due to exactly the same condition. If one entity is allowed to move, they are all potentially allowed to move. One can here, depending on the choice made under 3.3 below, either just move the **one at the front** of the list (provided the entities are sorted according to priority and entry time, so that front entity is the first one to move), or move **all** the entities, to the CEL. No search activity is then required. Examples of systems where all waiting appears to take place on separate delay lists are SLX, SIMAN, DEMOS and (Schriber & Brunner, 1997, p. 19, Schulze & Henriksen 1998, Banks *et alia* 1995 and Birtwistle 1979).

The efficiency of the two approaches is, however, contingent on the decision under 3.3, i.e. when to check for exit from the delay lists. If we at the same time are to check for many waiting conditions, it might be more efficient to scan just one single list than scanning many lists. There is always some overhead moving from list to list. If we only check for one specific waiting condition and the search is done in only one list, the search procedure is probably faster in the specific delay list than in the one general delay list. If the waiting of some entities will be ended by specific events, peculiar to this type of entity, while the end of waiting for many other entities has to be investigated together, there are hence reasons for a mixed approach, with some entities waiting on their specific delay lists, while other entities, although with different delay conditions, wait on a joint delay list. This will be discussed further below under 3.3.

The search of the joint list is inefficient only if there are **several** waiting conditions represented on this list. If one has only **one** type of condition on the list, the scan procedure could be very simple. Take the case when 100 entities wait on the common delay list, all with the same delay condition. Using the general procedure, which is necessary, if we do not know that the same condition applies to all entities, we might have to investigate the delay condition of all 100 entities, even if we search only for the first entity that can move.

Now suppose that the program has only one single waiting statement and that it is concerned with waiting that is to be handled on this joint single list. Assume further that this condition does not concern any value that can be specific for an entity, like a priority or other entity specific attribute (in GPSS: a parameter). In such a case, all waiting conditions on the joint list are certain to be the same. If we then find that the waiting condition is still true for the first entity on the list, then it is also true for the remaining 99 entities, which we then do not have to investigate. A procedure of this type has been introduced into micro-GPSS this year. In section 4, we show that this can speed up execution a great deal.

3.2 Types of delay lists used

In the case of a single delay list for all waiting entities, we distinguish between the case when there is a specific delay list for all conditionally delayed entities and the case when these delayed entities are on the **same** list as all the entities in the Ready State, i.e. when the single delay list and the CELs are merged into one single list. The first case can be exemplified by systems based on the three-phase approach and the second case by GPSS/H, where the Current Events Chain also incorporates all entities in the Condition Delayed State.

When it comes to **specific** delay lists, we can distinguish between the case when delay lists also perform the gathering of some queue statistics and the case when they serve only as a pure delay list. In the last case, additional linked lists are needed for the queue statistics gathering, e.g. to measure the time when an entity starts waiting. Examples of systems using the first approach are micro-GPSS and SIMAN (when using attached queues); an example of a system using the second approach is SLX (in its lowest layer form).

3.3 When shall the wait condition be tested?

We can here distinguish between three cases: **a**. The wait condition is tested after every single event or at least every event bringing the active entity to a stop. **b.** The wait condition is tested only after some very specific event has occurred. **c.** Some wait conditions are tested after every single event or entity stop event, while other wait conditions are tested only after some specific events have occurred.

The issue of when testing should take place is closely connected to the general issue, discussed above, of related or polled resolution of waiting. Related resolution for a specific condition implies that the test takes place in connection with the execution of a specific event, while polled resolution refers to the case when waiting is resolved at some other stage in the process. In case *a.* we use **polled** resolution , while we in case *b.* use **related** resolution for **every** delay condition. In case *c.* we use related resolution for some conditions and polled for others.

Combining the choices in this section with those in section 3.1, we find that the most efficient type of implementation from an execution point of view are the systems of type *Bb*, i.e. when there are as many delay lists as there are delay conditions and a wait condition is tested only after a specific event has occurred. In this case, the search of wait lists will take place only when certain events have taken place. In many programs, this concerns only a small subset of all events. Secondly, one does not have to scan through the whole list to find the entity to be moved first. One only needs to pick the entity at the front of the list. The search activity is hence minimal. The only example of this, at least for systems mentioned in this paper, appears to be SLX (with reactivation chains = delay lists; Henriksen 1995, p. 506). Since SLX is also compiled and skillfully programmed, it is very efficient not only in theory, but also in practice, as shown by various benchmark runs (Schulze and Preuss 1997).

The most **in**efficient system from a theoretical point of view is system *Aa*, i.e. there is only one single list for all entities waiting at wait statements and the wait condition is tested each time the active entity is stopped. Extensive scanning might take place after every few events. Since all waiting entities are on the same list, there is no guarantee that the entity that is to move, if any, will lie close to the head of the list. Sometimes, one might have to search through the whole list just to find the first one to move. An example of such a system is GPSS/H (see e.g. Schriber & Brunner, 1994, pp. 51-52).

This might be regarded as surprising, since GPSS/H is known to be relatively execution efficient. GPSS/H uses, however, for common simple waiting a mechanism that cuts down the search activity on the single list considerably, by each entity having a Scan Skip indicator. This indicator is switched on, if the entity's delay condition is true. At the search of the list, all entities having the indicator switched on can be skipped, which cuts down search time significantly. Another factor of importance is the fact that GPSS/H is a compiled system and skilfully programmed, using special procedures to speed up the hanling of waiting. This is an example of discrepancy between theoretical inefficiency and practical efficiency. Earlier GPSS versions, like GPSS/PC and GPSS V, using the same type of procedures as GPSS/H, display less discrepancy between the two efficiency concepts.

The question is then why do not more systems use the *Bb* combination, i.e. separate delay lists for every condition and only related waiting resolution. One reason is that related waiting is simple to implement **only** in the case of common simple waiting, referring mainly to servers and switches. It is then clear which events, in terms of operations, will send out the order to start looking at the delay condition, namely the events which refer to the start or end of using a server (like SEIZE and RELEASE) or to the setting or resetting of a switch. It is hence quite natural to have related waiting resolution in this case.

Already for the case of **special** simple waiting, it is difficult to implement related waiting, without including special instructions that a signal shall be sent out for these events. Take the case of waiting until the number of entities that has carried out a certain count has reached at least 4, with WAITIF N$LABEL<4 in micro-GPSS and TEST GE N$LABEL,4 in GPSS/H. The event that sends the signal can here be any kind of event. In order to have related resolution here, one must in principle define which events, or which control factors connected with events, will send signals. The difficulty becomes even larger in the case of complex waiting, dealing with more than one event.

In order to have related resolution also for special simple waiting and complex waiting, one must include information in the program that certain events are connected with related waiting. This is the approach taken by SLX. One here defines certain variables as control variables. Every operation that refers to such a control variable will send out a signal similar to that sent out by the events influencing common simple waiting in other systems. Thus, by defining in a program e.g. **control integer** *count* and **control boolean** *done*, one can have not only **wait until** *count*>10 and **wait until** *done*, but also **wait until** (*count*>5 **or** *done*). Every time an event dealing with a control defined value, like *count*, is carried out, a signal will be sent to all wait statements in which *count* is mentioned.

If the same control variable is used in only one or a couple of statements, the programming effort might not necessarily be significantly smaller than in some approaches, when all waiting resolution is solved by programmer supplied logic, like in DEMOS. Here one has to include a signal event after every event that can affect a conditional delay of the **special** simple waiting or complex waiting type. A potential problem with both of the SLX and DEMOS approaches, compared to a completely automatic approach, is that, if one forgets to define a variable as control, or to send a signal at its change, no checking will occur when this value changes. This can be a problem for inexperienced programmers.

The ideal would be to have related waiting for all events, without having to explicitly define control variables. I have investigated, if it would be possible in a system like micro-GPSS to have a pre-compilation phase in which all wait conditions are analyzed with the purpose of defining all variables that can occur in wait statements. My conclusion is that it would be impossible to define all such variables automatically. There are two main problems: 1. The use of parameters in indirect addressing referring to block numbers in a wait block, like WAITIF N(P1)>10. Since the entity dependent attribute P1 can take any integer value and N(P1) is the number of entities having reached the block with the number P1, all events have to be investigated, since any event can change the number of entities reaching a block. 2. The use of V$name in an expression, where the definition of this expression in turn refers to V$name1, which in turn refers to V$name2, etc.

Due to this, I have given up the idea of having automatic related resolution of all wait conditions. Related waiting resolution in micro-GPSS is hence limited to common simple waiting. The entities waiting due to other types of waiting conditions are placed on one single list, where the waiting resolution is handled by polling. It should be mentioned that there are in fact even some instances also of common simple waiting, where it is reasonable to use polled waiting, namely when multi-user servers (storages) are referred to by parameters with different conditions in the same program, like one condition P1=E (the storage with number P1 is empty) and later in the program another condition P2=F (the storage with number P2 is full). Since the parameters can take any value (up to the maximum number of servers allowed), it is problematic to implement separate lists for 1=E, 1=F, 2=E, 2=F,.., etc. In this case we only allow for related polling regarding P1=E, while the waiting regarding P2=F will be done on the one joint list with polled waiting resolution (Ståhl, 1996, p. 820).

3.4 How many entities shall simultaneously be removed from the delay list?

An important question is how to handle the move from a delay list to the CEL in detail. To exemplify, ten students wait on a delay list for books to arrive from the publisher, but only two books arrive. As the two books arrive, a signal (automatic or user supplied) is sent, implying that it is time to check the waiting condition. There are two main ways of handling this: 1. Let only **one** student at a time leave the delay list to go into the Ready State. 2. Let **all** waiting students leave the delay list at once.

If only one student moves at a time and we assume equal priority, the student having waited the longest time will move to the CEL. When the active entity, here e.g. the mailman having brought two books into the bookstore, has been brought to a halt, it is time for this first student entity to become the active entity. Having become active, it moves to the wait statement and it checks again that it does not have to wait any longer. If this is true, it can move on in the system. Before doing so, it will, however, since it comes from the delay list, cause the next entity, i.e. the student now in front on the delay list, to move to the CEL. The system must hence keep track of whether an entity comes to a wait operation for the first time or whether it comes from a delay list.

If, for some reason, all books have already been taken by some other entity, the now active entity will have to return to the delay list. (This will in the example happen for the third student.) If there are many entities on this list, it will require more computer time to sort it in the ordinary way from the end of the list, than just put it back on the first position of the list. However, in order to be sure that it is correct to put the entity back at the front of the list, the computer system must know for sure that there is no other entity that has been put back at the front of this list since the time when this entity was moved from the list. Such a system, avoiding time consuming sorting, is implemented in micro-GPSS.

Some other systems have **all** entities that wait on a delay list transferred at the same instant to the CEL. They are then, as they become active and find that the waiting condition holds again, placed, one at a time, back on the delay list, sorted in the ordinary order. In programs, in which a delay condition can switch back to being true again at the same value of the clock, like in the example with the bookstore, this might be an inefficient mechanism. The computer program of the simulation system can, however, be kept simpler.

3.5 Shall all waiting be explicit?

One design question is whether to have all waiting explicit. In the lowest, fundamental, level of SLX all waiting is explicit, using one single, but very efficient WAIT UNTIL statement, as discussed above. It appears also that some new systems, like e.g. Silk (Healy and Kilgore, 1997), have waiting explicit, with the waiting in front of a server taken care of by a wait operation and with the SEIZE block only handling the book-keeping regarding the utilization of the resource. In other systems, like GPSS/H, micro-GPSS and DEMOS, waiting in front of servers is implicit. The insertion of QUEUE blocks will not affect the actual waiting. In SIMAN one can use explicit queues, which function as delay lists, but if they are not used, implicit (internal) queues will be introduced automatically, e.g. in front of a SEIZE block. The actual execution might be different. Internal SIMAN queues, for example, always work with FIFO, in contrast to the explicit (attached) queues, where alternative priorities can be established. It appears, however, that as regards execution efficiency, the choice between implicit and explicit waiting is not as crucial as the other questions discussed above.

3.6 Shall waiting be in terms of WAIT UNTIL or WAIT WHILE?

Another choice to be made when constructing a simulation system, or a simulation model in a General Purpose Language, is whether the waiting condition shall be of the type WAIT UNTIL, implying that entities move from the delay list, when the condition becomes true, or of the WAIT WHILE type, implying that entities move from the list, when the condition becomes false. While most systems use a WAIT UNTIL statement, micro-GPSS (with WAITIF) uses a WAIT WHILE type of statement.

In micro-GPSS, we initially used the traditional GATE block of GPSS, which is of the WAIT UNTIL type, but around 1987 we switched to the WAITIF block, which works like a WAIT WHILE statement. We wait, prior to the WAITIF block, if, and as long as, the stated condition is **true**. The reason for the switch was partly pedagogical, partly one of ease of usage. The WAITIF statement is more in line with the IF block, where you go to an address, if the tested condition is **true**, in contrast to the traditional TEST block of GPSS (Ståhl 1993). Furthermore, we found that in around two thirds of our examples, WAIT WHILE applying to servers would allow a simpler code, like WAIT WHILE server is in Use (WAITIF server=U), than WAIT UNTIL server is **Not** in Use (GATE NU server). As regards execution efficiency, there is no measurable difference between the two approaches. It should finally be noted that in an extendable system, like SLX, a couple of lines of code are enough for implementing the WAIT WHILE statement.

4 THE SEQUENTIAL WAIT STATEMENTS

We study a method for more efficient execution of programs with complex wait conditions, implemented in micro-GPSS and dealing with complex wait conditions. We take the example in the introduction, namely with a ship that will wait before entering a harbor until there is no storm and there is a free berth as well as a free tug. We study the following micro-GPSS program.

It should be mentioned that the system on purpose is made to lead to explosive queueing so that one with a short program can illustrate the differences in execution times that can occur in longer programs. (For those more familiar with GPSS/H, the corresponding program in GPSS/H is presented in table 2 in section 5.)

We see at the top the following *valueof* expression: *notok valueof (tug=u)+(berth=f)+(storm=u).*

Table 1: Program 1 in micro-GPSS

```
        simulate
 notok  valueof   (tug=u)+(berth=f)+(storm=u)
 notok2 valueof   (tug=u)+(storm=u)
 berth  storage   3
*  Storm segment
        generate  ,,,1
 next   advance   48*fn$xpdis
        seize     storm
        advance   4,2
        release   storm
        goto      next
* Ship segment
        generate  7,5
        arrive    area
        waitif    v$notok>0
        seize     tug
        enter     berth
        advance   2
        release   tug
        advance   18,4
        waitif    v$notok2>0
        seize     tug
        advance   2
        release   tug
        leave     berth
        depart    area
        terminate
*  Stop segment
        generate  14050
        terminate 1
        start     1
        end
```

v$notok will take a value >0, if at least one of the three conditions is true, i.e. the tug is in use, the berth is full or there is a storm going on. The ship will wait as long as this is true. Since this is clearly a complex waiting condition, the waiting will be resolved by the polled method, with all ships waiting on the joint delay list, investigated after each event. This causes the program example to execute quite slowly. On a 233 MHz Pentium the run time was 140 seconds.

In the program we also have a similar wait condition, when the boats leave the harbor, now with the difference that they only have to wait for a tug and for the storm to stop. Replacing *waitif v$notok2>0* with the two sequential blocks *waitif storm=u* and *waitif tug=u,1,* we can speed up the program execution greatly. The main reason for the speed up is the C operand 1 of the block *waitif tug=u,1.* This number 1 implies that there is one, immediately preceding, WAITIF block connected with this block, which the ship entity must have gone through at exactly the same clock time, in order to pass through this last block. Assume the ship goes through *waitif storm=u* at time 200, but there is no tug then, and the tug is free first at time 210. When the ship at time 210 is able to pass through the block *waitif tug=u,1,* it will then have to go back to the block *waitif storm=u* and check that it can now go through both blocks at the same time. This ensures that the ship will leave the harbor only when at the **same** time there is a tug and no storm (Ståhl 1990, p. 332).

Each simple sequential WAITIF block has its own specific delay list, subject to related resolution. We test for delay resolution only when there is an event affecting STORM or TUG. When we ran this modified program with the sequential WAITIF blocks on the mentioned computer, it took only 7 seconds. The original program hence took 20 times longer time to execute.

We can also replace the block *waitif v$notok>0* with the three blocks *waitif storm=u, waitif berth=f* and *waitif tug=u,2.* Here we will from *waitif tug=u,2* go back two blocks to *waitif storm=u,* if we have not gone through the last of these three blocks at the same time as we were able to pass through the first of these blocks. If we run this modified program, we will find that it will not run much faster than the one, where we had eliminated only the block *waitif v$notok2>0.* The reason for this has to do with the facts discussed at the end of section 3.1. By doing the first elimination, the block *waitif v$notok>0* became the **only** waiting block and hence all conditions on the joint list became identical and it is enough to look at the very first entity on the list.

5 THE BORN SANDWICH

If one has a good understanding of how various wait conditions work, the simulation modeler can in certain cases design the program so that it will run a lot more efficiently than would otherwise be the case. We shall here introduce only one such design feature, which can be important in not only micro-GPSS, but also in several other simulation systems.

We shall call this the Born sandwich after Professor R. Born, who introduced this for the first time in an article this year, applying it to a micro-GPSS program (Born 1998, pp. 315-316). The idea is quite simple, but I have not found any similar idea in e.g. the standard GPSS textbooks. It involves putting a SEIZE block on top of the wait block and a RELEASE block below the wait block, thus constituting a kind of sandwich.

The idea is that when a wait statement executes very slowly, because one has to scan a great number of entities on the delay list, and the scanning of each activity takes noticeable time, then one can cut down the search activity by putting a SEIZE block in front of the wait block and a RELEASE block immediately after this. Only one entity will then be allowed to rest between the SEIZE and RELEASE blocks. Hence, only one entity at a time will be waiting on the joint delay list, on which search is slow due to polled waiting resolution. All entities instead wait on a delay list with related resolution of waiting and hence faster search.

When I used a Born sandwich in program 1, by inserting *seize wait* before, and *release wait* after, the block *waitif v$notok>0* and *seize wait2* before, and *release wait2* after, the block *waitif v$notok2>0,* the execution time went down from 140 to 10 seconds.

Table 2: Program 1 in GPSS/H

```
       SIMULATE
 ITSOK  BVARIABLE  FNU$TUG*SNF$BERTH*FNU$STORM
 ITSOK2 BVARIABLE  FNU$TUG*FNU$STORM
 BERTH  STORAGE    3
* Storm segment
       GENERATE   ,,,1
 NEXT   ADVANCE    48*RVEXPO(2,1)
        SEIZE      STORM
       ADVANCE    4,2
       RELEASE    STORM
       TRANSFER   ,NEXT
* Ship segment
       GENERATE   7,5
       QUEUE      AREA
       TEST E     BV$ITSOK,1
       SEIZE      TUG
       ENTER      BERTH
       ADVANCE    2
       RELEASE    TUG
       ADVANCE    18,4
       TEST E     BV$ITSOK2,1
       SEIZE      TUG
       ADVANCE    2
       RELEASE    TUG
       LEAVE      BERTH
       DEPART     AREA
       TERMINATE
* Stop segment
       GENERATE   14050
       TERMINATE  1
       START      1
       END
```

It should finally be shown that this Born sandwich idea can be useful also in other simulation systems than micro-GPSS, in fact in any system, where related resolution of waiting is used for common simple waiting and polled resolution of waiting is used for other types of waiting. Then the introduction of common simple waiting, referring to a server, with a capacity of only **one** entity, around a statement referring to another type of waiting, implies that there will be at most only one single entity waiting on the delay list, on which search would be very slow, when there are many entities waiting. The Born sandwich can be used, for example, in the GPSS/H program on the next page, which is a translation of program 1 above. The Born sandwich will here involve the two TEST blocks.

There is also here a significant reduction in execution time, although not of the same magnitude as for the program in table 1. It should be mentioned that one in GPSS/H, instead of *SEIZE WAIT*, *TEST E BV$ITSOK,1* and *RELEASE WAIT*, as an alternative could write *LINK WAIT,FIFO,TOK*, *TOK TEST E BV$ITSOK,1* and *UNLINK WAIT,TOK,1*. This is even more efficient. We can call this a Crain sandwich after its inventor R. Crain.

ACKNOWLEDGEMENTS

The author gratefully acknowledges valuable comments from T. J. Schriber, T. Schulze and J. O. Henriksen.

REFERENCES

Banks, J., B. Burnette, H. Kozloski and J. Rose, 1995, *Introduction to SIMAN V and Cinema V*, Wiley, New York.

Birtwistle, G.M., 1979, *Discrete Event Modelling on Simula*, Macmillan, London.

Born, R. G., 1998, Teaching Simulation of Manufacturing Systems Through Stepwise Refinement Using micro-GPSS, in D. Davani and D. Elizandro (eds.) *International Conference on Simulation and Multimedia in Engineering Education (ICSEE '98)*, SCS, La Jolla.

Davis, R. and R. O'Keefe, 1989, *Simulation Modelling with Pascal*, Prentice Hall, New York.

Healy, K. and R. Kilgore, 1997, Silk^{TM} H: A Java-Based Process Simulation Language. In S. Andradóttir, K. Healey, D. Withers and B. Nelson (eds.), *Proceedings of the 1997 Winter Simulation Conference*, SCS.

Henriksen, J.O., 1995, An Introduction to SLX. In C. Alexopolos, K. Kang, W.R. Lilegdon and D. Goldsman (eds.) *Proceedings of the 1995 Winter Simulation Conference*, SCS.

Schriber, T. J. and D. T. Brunner, 1994. Inside Simulation Software: How It works and Why It Matters. In J. Tew, S. Manivannan, D. Sadowski, and A. Seila (eds.), *Proceedings of the 1994 Winter Simulation Conference*, SCS.

Schriber, T. J. and D. T. Brunner, 1997. Inside Simulation Software: How It works and Why It Matters. In S. Andradóttir, K. Healey, D. Withers and B. Nelson (eds.), *Proceedings of the 1997 Winter Simulation Conference*, SCS.

Schulze, T. and J. Henriksen, 1998, *Simulation Needs SLX*, Otto-von-Guericke Universität, Magdeburg.

Schulze, T. and F. Preuss, 1997, Benchmarks für diskrete Simulationssysteme. In Deussen, O. and P. Lorenz (eds.), *Simulation und Animation '97*, SCS, Erlangen.

Ståhl, I., 1990, *Introduction to Simulation with GPSS: On the PC, Macintosh and VAX*, Prentice Hall International, Hemel Hempstead, U.K., 1990.

Ståhl, I., 1993, Principles Behind the Design of an Easy-to-Learn Simulation Language. In Roberts, R. and S. Monroe (eds.), *Simulation Applications in Business Management and MIS*, SCS, La Jolla.

Ståhl, I., 1996, Steps towards a Better Internal GPSS Mechanism. In J. Charnes, D. Morrice, D. Brunner and J. Swain (eds.), *Proceedings of the 1996 Winter Simulation Conference*, SCS.

Vaucher, J., 1977, Code of GPSSS, version 5.1, at http://www.jsp.umontreal.ca/~vaucher/Software/gpsss.sim

AUTHOR BIOGRAPHY

INGOLF STÅHL is Professor at the Stockholm School of Economics, Stockholm, and has a chair in Computer Based Applications of Economic Theory. He was visiting Professor, Hofstra University, N.Y., 1983-1985 and leader of a research project on interactive simulation at the International Institute for Applied Systems Analysis, Vienna, 1979-1982. He has taught GPSS for twenty years to over 5000 students at universities and colleges in Sweden and the USA. He has on the basis of this experience led the development of the micro-GPSS system. He is now involved in putting the micro-GPSS system with a tutorial on the Web.