Adapting Synchronous Parallielism to AI Scripting in Games

Joseph Kehoe[1], Joseph Morris[2]
[1]Institute of Technology
Carlow
[2]Dublin City University. Dublin, Ireland
joseph.kehoe@ITCarlow.ie, joseph.morris@computing.dcu.ie

ABSTRACT: *Scripting is used extensively in game development for the implementation of AI based behaviors. Game scripting languages have not, as yet, benefited from the move to multicore architectures. We propose a synchronous model of concurrency particular to game AI scripting that allows game scripting languages to fully utilize multicore processors. The next steps in developing this model further are discussed.*

Keywords: AI Scripting, Intelligent Games, Games Architecture, Multicore processors

Received: 19 January 2012, Revised 1 March 2012, Accepted 9 March 2012

1. Introduction

Processor power (and speed) has been increasing at an almost constant rate, following Moore's law, since the introduction of the integrated circuit. Up until recently this increase in speed has been attained through greater miniaturization and complexity of single processor chips. Future increases in speed will be through the use of multiple core processors, [2] where more speed will mean more cores on each processor. Processors which used to contain a single complex core will now be replaced by processors containing many cores [11], [3].

The only way for software to take advantage of these new architectures is by simultaneously using as many of the cores as possible. Unfortunately, writing concurrent code is more difficult than writing sequential code. New techniques for designing, writing and testing concurrent software are needed.

Synchronous languages are the technology of choice for real time embedded systems. They have a solid mathematical foundation and promise concurrency, simplicity and synchrony [1]. By providing synchrony they remove the nondeterminism that comes with most other concurrency models. Their division of time into discrete instants corresponds closely with frame based approach of computer games. Yet despite this, synchronous languages are not used in computer game development. We believe it is possible to adapt the synchronous model to game scripting.

AI scripting is an integral part of computer game development. Scripting is essential for two reasons, the nature of the game development process and the type of game developers who write game behaviors.

The game development process is an iterative one. This is particularly true for entity behavior design and implementation. Entities in games are any objects that can interact with their surroundings and the player of the game. They range from pretty trivial items such as doors (that can be locked, unlocked, opened and closed) to non player characters that can form their own plans. Game entity behaviors can really only be properly assessed by actually playing the game with the entity behaviors *in situ*, thus leading to the employment of rapid *design-implement-playtest* iterations. For this to be a viable process each iteration needs to be as short as possible. It can take many iterations before a behavior becomes acceptable within a particular game. These rapid iterations presume the use of scripting languages, i.e. high level languages that allow for rapid implementation (or prototyping) of behaviors.

In this paper we propose a model of concurrency developed specifically for game scripting. Our model will allow game scripting languages to fully utilize multicore processors while still remaining simple enough for game developers to use. We get the benefit of multiple cores without the overhead of programming complexity associated with concurrent programming.

1.1 Overview of Paper

In the next section we examine the game problem domain. In particular we look at the software process associated with video game development. From this we identify the important and defining properties unique to this area. Any model of concurrency proposed specifically for game development needs to be constrained by this list of properties. We then look at the synchronous approach to concurrency and identify the reasons for its non usage in game development. Then in the next section we give an overview of the model we propose. We look at key features of this model of concurrency and show how it fulfills the expectations set out in the previous section. We follow by reviewing related work in concurrency and game scripting. Finally we finish with our conclusions and list further work that remains to be completed.

2. The Game Scripting Problem Domain

Here we describe the characteristics of this domain in terms of the general architecture used in game software and in terms of the development process that is used to develop these games. We can then use these properties to judge whether our proposed model is an acceptable one.

2.1 Video Game Architecture

Video games tend to follow a standard software architecture. This architecture is based on two properties of video games.

1. Video games are simulations;
2. Video games are frame based.

The second property reflects on how video game simulations are run. A frame represents an instant in time. Game speed is determined by how many instants it can compute every second, a metric known as the frame rate. Each frame represents a snapshot of the game world at an instant in time. To compute this snapshot the game must execute the following steps:

1. Gather Input;
2. Run Physics;
3. Send/Receive Network Messages;
4. Run AI/Behaviors;
5. Update Game State;
6. Render Scene.

Because each frame represents a moment in time the order in which these steps are executed is of no importance except for the rendering of the scene and updating of game state. The rendering of the scene must be the last step because we cannot render the scene correctly until all the other steps have completed. We can consider all of the other steps to be simultaneous in time.

The main body of a video game is the computing of frames at between some set minimum and maximum frame rates. This puts hard time constraints on each game. All steps for computation of a frame must be completed within their allotted time.

Behavior code is generally allocated only a small fraction of the time available within each frame. All behavior scripts must then be guaranteed to complete within this allotted time.

2.2 Video Game Development Process
The video game development model has some characteristic features that must be taken into account. Specifically we are concerned with those aspects that impinge on the AI or entity behavior programming.

Game entity behavior is developed using an evolutionary software development process. Each iteration of development is interleaved with play testing. This play testing is frequent and each individual behavior has to be play tested. Play testing is the only way of ensuring that a behavior works in a game because each behavior has to be seen in context. As a result play testing forms an essential part of game development.

Behaviors are attached to game entities. Entities range from complex non player characters (NPC's) to simple objects such as doors. The majorities of behaviors are simple both in terms of the nature of the code implementing the behaviors and the amount of code required. Because games are, in essence, simulation programs an object oriented or object based paradigm maps nicely to the problem domain and provides a easy way to break code into discrete independent chunks.

Behaviors are implemented by scripters who range in programming ability from professional programmers to hobbyists with only a basic understanding of programming. Even if we were to exclude hobbyists the scripting process must be accessible to game designers as well as game programmers. Game designers decide on what the appropriate behaviors are for each entity. They need to be able to directly implement as many of these themselves as is possible. While the more complex NPC AI routines might be left to dedicated AI programmers most of the entities in a game operate at a much simpler level. For example, the behavior of a door should not require a dedicated professional AI programer to implement. Ideally, the game designer herself, as the person who fully understands the required behavior, should be able to implement the behaviors directly. To make this possible a simple high level scripting language is required.

2.3 Required Concurrency Model Properties
From the previous discussion we can see that any proposed model of concurrency must have the following properties to be considered plausible in this problem domain.

2.3.1 Simulation Based
The model should map naturally onto the object based paradigm used in simulations. The closer the fit the easier it will be to incorporate into the games domain.

2.3.2 High Level
The model should not involve the user (scripter) in the low level details of the implementation of concurrency. It should also be easily incorporated into existing high level scripting languages.

2.3.3 Predictable
The model should, as far as possible, allow for predictable run times so that guarantees can be given as to the amount of time required to process the behavior generating scripts during each frame.

2.3.4 No Deadlock or Livelock
The model should remove deadlock and livelock. Reasoning about deadlock and livelock requires a high level of programming skill that scripters cannot be assumed to have. The user of the model, the scripter, should not have to worry about these issues.

3. Synchronous Parallelism

Synchronous languages are based on a common mathematical framework. In this framework time advances in lockstep with one or more clocks. This lockstep gives the synchronous model the property of determinism. Determinism ensures that behaviors of synchronous systems are reproducible thus simplifying the testing and debugging of those systems. It is no surprise then that the synchronous approach is prevalent in the development of safety critical systems where not only must the implementation be correct but users must be convinced that it is correct. Synchronous languages help foster confidence in the correctness of the proposed solution but also help keep development and maintenance costs down [1].

The synchronous model supports the following three properties:

1. Support of Concurrency in a user friendly way;
2. Simplicity of the underlying model;
3. Synchrony, supporting the simple and frequently used implementations (e.g. with finite memory and time).

Synchronous languages divide time into discrete instants and a program progresses through successive *atomic reactions*. This approach is most commonly used by control engineers and hardware designers where correctness is important and the underlying implementation (digital circuits) are, in the main, synchronous in nature.

The best known synchronous languages are VHDL, Lustre, Esterel and Signal each of which takes a slightly different approach. Of these, Esterel [4] is probably the most familiar to software developers. Esterel sports a traditional imperative syntax and has been integrated into C (ECL) and Java (Jester). It has also been integrated into the most widely used OO modeling language UML. Under Esterel reactions are instantaneous (atomic). It is this property that gives it determinism. Communication between different "*threads*" is through signals, which are considered instantaneous. In fact several emissions and receptions of signals can occur in sequence within the same instant.

Synchronous languages, when compiled to code, are generally compiled into sequential (single threaded) code. In Esterel this is usually through transformation of the specification into a finite automata. This code tends to be inefficient but recent work has shown that it is possible to transform specifications into multithreaded code using OpenMP [14].

The synchronous approach has been embraced by the hardware community but not the software community. There are a number of reasons for this. Firstly, synchronous languages are high level formal specification languages that use formalisms more familiar to electrical engineers and hardware designers than most software developers. In particular, these formalisms are unsuitable for the game scripting community many of whom have no formal CS qualifications. The specification languages themselves are explicitly concurrent so although they provide determinism they do not hide the concurrency from the person who writes the specification.

Secondly, there is the difficulty in producing efficient parallel code from these specifications. Until recently the compiled implementations were sequential and although there has been recent work [4], [14] in this area it remains preliminary in nature. Benchmarks are either sparse or give mixed results. This may be because the level of granularity provided by these implementations is not be optimal.

Finally, the synchronous model should ideally be integrate-able into existing scripting languages. A switch from using scripting languages to formal specifications does not fit in with the evolutionary model employed in game AI development.

Synchronous languages offer much and the underlying model matches the structure of game software. What is needed is a way of adapting this model into the game development process. We propose a synchronous model that addresses all these issues and is suitable for AI development in games.

4. Proposed Concurrency Model

A game is a simulation that consists of a set of entities interacting with each other in some world. Each entity has its own state and a set of behaviors that determines how it responds to various stimuli. The overall state of the game is given by the state of the entities that it contains. Games proceed in a stepwise fashion as a series of discrete moments in time. At each step the entities update their state based on the events or stimuli generated during the previous step. Overall entity behavior through the lifetime of a game comprises the sequence of step behaviors made during that game. Concurrency occurs naturally at the entity level.

4.1 Entities
Game entities can be divided into two broad categories: Reactive and Active. Reactive entities act only in response to some externally generated stimulus. They tend to have only simple behaviors and usually represent items such as doors or guns.

Active Entities respond to externally generated stimuli as well but also contain purposeful behavior that does not depend on external stimulus. an active entity is guaranteed to perform some behavior during every step whether it receives any external stimulus or not. It is as if an internal stimulus is generated by the entity itself at the start of (or maybe even caused by) every step.

Active entities have more complex behaviors than reactive entities with their behaviors generally based on well known AI algorithms. In a game active entities model either non-player characters (NPC's) or the players themselves. While NPC behaviors are based on AI algorithms, internal player behavior is based on the continuous input from a peripheral device, such as a gamepad, operated by the player.

An entity is composed of four components. These are State, Interface, Constraints and Message Queue.

State is a set of named attributes with associated values for each. A subset of these attributes will be immutable. Immutable attributes are given values on entity creation and remain unchanged until entity destruction. The full set of an entities named attributes is called the entity state.

Secondly, a set of acceptable messages. Each message represents the response of the entity to a specific stimulus. The set of acceptable messages is known as the entity interface. An entity can only respond to a stimulus if it has a corresponding message defined in its interface.

Thirdly, a constraint list. Entity attributes may be subject to local and global constraints. The set of local constraints determines the set of acceptable combinations of values that the attributes can hold. The global constraint set determines allowable combinations of entity state values that different entities can simultaneously hold.

Finally the message queue contains the full set of outstanding messages that the entity has to respond to during the current step. This queue will contain all messages generated for the entity during the previous step.

Each entity will contain at least two attributes as part of its state. These are the entity ID which uniquely identifies the entity and the entity TypeID which identifies the type of the entity. Both of these are immutable attributes. Every entity that has the same state, interface and constraint set will have the same type.

Entities know a non empty subset of the state belonging to any other entity. This subset will contain, at a minimum, the entity ID and TypeID attributes. Entities know the full interface of every other entity. If an entity knows the ttribute of another entity then it is allowed to read the value contained by that attribute but it cannot write to a attribute in any other entities state. An entity is allowed send a message to any entity if it knows the value of that entities ID attribute.

4.2 Messages

A message consists of:

1. The sender ID;
2. The receiver ID;
3. A collection of attributes for the entity and the values of those attributes.

A message generated (or sent) during one step will always be received during the next step. In response to receiving a message an entity can do the following:

1. Send messages to other entities whose ID value it knows;
2. Create a new entity or entities;
3. Update any of its own mutable attributes provided that these updates do not violate any constraints in its constraint set;
4. Destroy itself.

An entity can only respond to a message defined in its interface. All messages sent to an entity that are not defined in its interface are ignored.

4.3 Steps

A complete computation consists of a sequence of two or more steps. During the first step initialisation occurs where:

1. Entities are created;
2. Initial Messages generated.

During the final step all remaining entities are destroyed. For every other step all messages that were generated during the previous step are processed. Local state is updated instantaneously and simultaneously at the end of each step. State update is defined as the sequential composition of the messages contained in the message queue, in order, modified by the conflict resolver. The conflict resolver is any well defined algorithm that ensures that any state update does not violate any constraints in the entities constraint list.

Local conflicts can be simply resolved by discarding any messages that cause conflicts. Global conflicts, however, can only be detected once the step is completed so a state rollback of any entity state that causes a conflict will be needed.

Any messages generated during this process are delivered instantaneously at step end. Delivered messages are put in the receivers message queue in an order defined by the message sorting algorithm. The message sorting algorithm an be any algorithm that guarantees that:

1. Messages generated by a single entity for the same receiver are placed in the message queue in the same order that they were generated in;

2. The final order of the message queue is deterministic. That is, for any given set of messages their ordering is unique and will always be the same regardless of how many times the ordering algorithm is applied.

4.4 Model Properties

Our model has a number of interesting and useful properties. Here we outline some of them.

Game scripters cannot be assumed to be technically adept programmers. It is common and accepted practice to use an object based approach in game scripting because it maps so naturally to simulation problems. Our approach, through the use of entities, implicitly defines the concurrency in an application. The identification of entities within a game, something that scripters already do, automatically divides the code into independent concurrent pieces. This proposed model allows them to produce concurrent code without requiring that they be aware of how that concurrency is achieved. Each messages can only write to attributes belonging to the receiving entity but can read attributes belonging to any other entity. During each step all entities can operate in tandem. As each entity has its own thread of control we can guarantee that for each entity only one of its messages can be executing at any one time. Therefore as an entities attributes can only be written to by one of its own messages it automatically follows that we do not need a write lock when updating an attribute.

When a message reads the value of an attribute belonging to another entity it can only read the public value. The public value is immutable during a step so no read locks are required either. Since no locks are required and all messages are asynchronous there can be no waiting.

At the start of each step there is a predetermined amount of work to do. This work comes in the form of the message queues. During the step this workload is not added to. Any newly generated messages are held until the next step and will form the workload for that step. We have already seen that there is no waiting during a step. This means that it is possible, at the start of each step, to determine the amount of work to be done and the total computation time required.

This model ensures that the outcome of any computation is independent of the underlying parallelism employed. The same computation, when run on different systems, should always give the same outcome. Given that we start with the same initial conditions we need to show that each step produces the same results. We can do this inductively. For the base case we know that the initial conditions are fixed at the start of step 0. We also need to show that the state of the entities at the end of the step will always be the same.

Taking each entity in turn we can see that the final state is dependent on

1. Order of Messages processed during step;

2. Initial State of Entity;

3. State of any other entities read during step.

We can assume, using our inductive hypothesis, that the initial state of the entities and the content of, and ordering in, the message queues are fixed. During message execution we can only access the public state of other entities and these are immutable throughout a step and equal to the initial state. Therefore the computation must always proceed in the same manner and each entity proceeds independently of all other entities.

At the end of the step each entity will have produced a set of messages for the next step. The model requires that the message queues are ordered deterministically. That is, there is only one correct ordering for any given set of messages. If, at the end of a step, we have produced a set of messages then the order in which they were produced is immaterial because they can be sorted into only one possible ordering for the next step.

Thus if the conditions at the start of the step were fixed then the results at the end will be fixed. This shows that computation always proceeds in a fixed manner independently of the underlying amount of parallelism. The built in determinism also means that debugging of code is not made more difficult by the introduction of concurrency.

5. Related Work

5.1 BSP - Bulk Synchronous Processing

BSP has been proposed as a bridging model for general purpose parallel compuation by [13]. A BSP computation consists of a sequence of supersteps. In a super-step each component (a processor or core) is allocated a task consisting of a combination of local computation and, message transmission and reception from other components. After L time units have passed a check is made to see if the super-step has completed. If it has, then the next super-step is started. Otherwise the next L units are allocated to completing the current super-step.

In simple terms, at each step a set of local computations is undertaken concurrently. The aims of BSP are to make it simple to write concurrent code, be independent of target architectures and make performance of a program on a given architecture predictable [12]. BSP allows you to put an upper time bound on a computation for a particular architecture. This makes the performance more predictable. In addition, deadlock using BSP is impossible. BSP is also easier to debug in that computations can be rearranged inside a superstep without affecting the outcome. BSP has been successfully integrated in scripting languages in the past by, for example, [10].

Our proposed model bears similarity to the BSP approach. Amongst the differences is our placing of concurrency at the object level, thus making it an implicit part of the object model.

^

5.2 COOP - Concurrent Object Oriented Processing

The most popular form of concurrent object oriented programming model is based on active objects. An active object is an object that runs inside its own thread. It represents a merging of process and object [6], [9]. The internal state of an active object is private to that object. Any interaction that must take place between objects must take place via message passing. Generally, messages are asynchronous but there is variation between explicit or implicit acceptance of messages by objects.

Actor models of concurrency are closely based on active objects. An Actor is an active object that can send finite set of messages to other actors, create a finite set of new actors and define how it will behave in relation to the next incoming message. [7]

Combining active objects with the synchronous model simplifies the object model from the programmer point of view, in particular by allowing direct read access to objects.

6. Conclusion

We have outlined a model of concurrency developed specifically for AI development in games. AI scripting is undertaken by game designers who are not professional programmers and therefore do not have an in-depth understanding of concurrency. As the game entity behaviors they develop have to be play tested to ensure they are appropriate they must use many rapid *design-implement-playtest* iterations during development. To enable them to make use of concurrency we developed a model that is easy to use, removes as much of the burden from the designer as possible while retaining the benefits promised by the synchronous approach.

6.1 Remaining Work

Some work remains to be done on developing algorithms that can be used by the conflict resolver. Although a simple conflict resolution algorithm has been proposed it may be the case that different games will need to use different or more sophisticated conflict resolving algorithms.

The next stage of this work is the production of an implementation of our model to demonstrate the viability of this approach. It will be incorporated into an existing well known game scripting language, Lua [8], to show how it fits into the development tools and practices used in the games industry. This will also serve to show how transparent this model is to game scripters in practice.

References

[1] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R. (2003). The synchronous languages 12 years later. *In*: Proceedings of the IEEE, 91 (1) 64 - 83.

[2] Blake, G., Dreslinski, R., Mudge, T. (2009). A survey of multicore processors. *Signal Processing Magazine*, IEEE, 26 (6) 26-37, november. .

[3] Borkar, S., Mulder, H., Dubey, P., Pawlowski, S., Kahn, K., Rattner, J., Kuck, D. (2006). Platform 2015: Intel processor and platform evolution for the next decade.

[4] Boussinot, F., de Simone, R. (1991). The esterel language. *In*: Proceedings of the IEEE, 79 (9)1293 -1304.

[5] Brandt, J., Schneider, K., Shukla, S. K. (2010). Translating concurrent action oriented specifications to synchronous guarded actions. SIGPLAN Not., 45, 47 - 56.

[6] Briot, J. P., Guerraoui, R., Lohr, K. P. (1998). Concurrency and distribution in object-oriented programming. ACM Comput. Surv., 30 (3) 291 - 329.

[7] Correa, F. (2009). Actors in a new "*highly parallel*" world. In WUP '09: *In*: Proceedings of the Warm Up Workshop for ACM/ IEEE ICSE, p. 21- 24, New York, NY, USA. ACM.

[8] de Figueiredo, L. H., Celes, W., Ierusalimschy, R. (1996). Lua - an extensible extension language. Software: Practice & Experience, 26, 635 - 652.

[9] Hernandez, J., de Miguel, P., Barrena, M., Martinez, J., Polo, A., Nieto, M. (1994). Parallel and distributed programming with an actor-based language. *In*: Proceedings of the Second Euromicro Workshop on Parallel and Distributed Processing., p. 420-427, 26-28 .

[10] Hinsen, K. (2007). Parallel scripting with python. *Computing in Science and Engineering,* 9 (6) 82-89.

[11] Held, J. B. J., Koehl, S. (2006). From a few cores to many: A tera scale computing research review. Intel White Paper.

[12] Skillicorn, D. B., Talia, D. (1998). Models and languages for parallel computation. *ACM Comput. Surv.*, 30 (2) 123-169.

[13] Valiant, L. G. (1990). A bridging model for parallel computation. Commun. ACM, 33 (8)103-111.

[14] Yuan, S., Yoong, L. H., Roop, P. (2011). Compiling esterel for multi-core execution. *In*: Digital System Design (DSD), 2011 14th Euromicro Conference on, p. 727 -735, 31.