COROUTINE SYNCHRONIZATION IN AVS

Gudrun J. Klinker
Cambridge Research Lab, Digital Equipment Corporation
One Kendall Square, Cambridge, MA 02139

Abstract

The current AVS flow executive provides only limited mechanisms for coroutines synchronization: either they run completely synchronously with the entire network or completely asynchronously. In many real-time applications, responding to user interaction or to real-time sensors (live camera input), finer-grained synchronization control is needed. This paper presents a token-based handshaking scheme which can be instantiated at runtime between any subgroups of modules and coroutines, allowing users to define control flow in the network, as well as data flow.

Developer's Track,
Third International AVS User Conference (AVS 94),
Boston, MA, May 2-4, 1994.

INTRODUCTION

Interactive data exploration and event monitoring systems need special, continuously running processes to accept asynchronous input from user interaction [4, 5] and remote sensors[1]. AVS provides *coroutines* for this purpose. They complement the more commonly used *modules* in AVS data flow networks. AVS coroutines execute continously as independent processes. In contrast, modules are scheduled by the AVS flow executive only when their input data or parameters have changed.

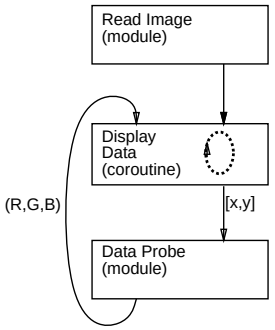

Figure 1: Data probing application

Figure 1 shows a simple data probing task involving a coroutine and a module. The “Display Data” coroutine repeatedly polls for new cursor positions. When the user moves the mouse, the coroutine forwards the mouse coordinates to a “Data Probe” module to determine the data values of the selected pixel. More sophisticated data exploration tasks can be generated by replacing the Data Probe with other modules, e.g., image cropping, filtering, defining a seed point for region growing, and computation of statistical information in a local image area.

For event monitoring applications, fast response from coroutines and dependent modules is extremely critical. This goal can be impossible to achieve when running a compute-intensive application that also requires continual user input. When users rapidly move the mouse, application developers need to decide whether it is better to finish the current computational unit before reacting to newer user input or to abort the current module execution and restart.

In AVS, coroutines run either synchronously or asynchronously with the data flow network. In the asynchronous case, coroutines execute continuously and send out data at their own pace. Whenever new data becomes available, AVS interrupts the execution of connected modules downstream, restarting them on the new data. Modules may never finish their execution if the coroutine produces new data faster than the dependent modules can process it. This can be the case during periods of fast user interaction, leading to a complete lack of response – and even more furious user interaction. In the synchronous case, AVS assures that the coroutine halts until all modules have finished their computation. Unfortunately, only one coroutine can be synchronized this way with the data flow network [1]. Many applications need more than one coroutine, e.g. to obtain live video input and interact with it, or interact with two images using linked cursors and two Display Data coroutines [4, 5]. And even with a single coroutine in a network, users may prefer specifying a suitable subset of critical modules over the all-or-nothing approach currently provided by AVS.

These observations indicate that application developers need tools to define appropriate, application dependent scheduling priorities. This paper presents a handshaking protocol for coroutines and modules, using token flow to specify module and coroutine execution patterns in AVS data flow networks.

[1] as well as to automatically generate data in animation and simulation applications

INTERPLAY BETWEEN A COROUTINE AND ONE OR MORE MODULES

In many cases, it is important for interactive applications to provide scheduling constraints between a coroutine and a selected subset of modules, so that the coroutine waits for this set of modules to finish before it sends out more data.

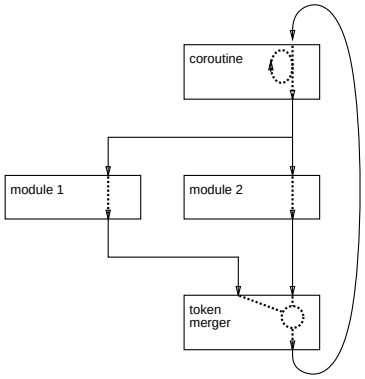

Figure 2: Control flow between one coroutine and two modules

Figure 2 shows a token flow scheme with which developers can specify – at runtime – how a coroutine should interplay with several parallel modules.[2] When the coroutine sends out new information, it also sends a new token and idles until it receives it back. Each module passes the new token on when it is finished. A Token Merger can use various strategies for collecting duplicate copies of a token from several parallel modules and returning them to the coroutine: it can wait 1) until it has received all tokens from all modules or 2) until the token from the first finished module arrives. Another strategy 3) allows users to specify that only tokens from one specific module should be forwarded. The module can either ignore further copies of the same token from the other modules, or it can forward those all as well. Users can switch between strategies at runtime, using the AVS parameter panel.

Figure 3 provides skeletons of the coroutine and module code that handle the token flow. When the coroutine receives new input (steps 1, 2), it determines whether the current input token matches the last output token (step 3). If the tokens match, the coroutine processes the new input (step 4), generating a new output token and sending out all appropriate information (steps 5, 6). If the input and output tokens are not the same, the coroutine saves parameter and input changes (steps 7, 8) – so that they will not be skipped – and resumes its waiting position (step 1). In the meantime, connected modules receive and forward the new token, as shown in the module skeleton.

Several details are crucial to the robustness of the token passing scheme. Zero-valued input tokens occur when the token input port is not connected, i.e., when the user does not want to use the handshaking scheme. The coroutine skeleton treats such intokens as a special case (step 3), allowing them to bypass the token checking scheme. Furthermore, the token flow can become misaligned when developers build or change the data flow network, bringing handshakes between modules and coroutines to an unexpected halt. Users can jumpstart the system, using the “go_on” parameter.

[2] The figure shows only the token flow; other data flow must be added in real applications.

```
/*
/* Skeleton coroutine to process token flow
*/
corout_description () {
    AVScreate_input_port (‘‘my_intoken’’, ‘‘integer’’, OPTIONAL);
    AVScreate_output_port (‘‘my_outtoken’’, ‘‘integer’’);
    AVScreate_parameter (‘‘go on’’, ‘‘oneshot’’, 0,0,0);
}
main (int argc, char *argv[]) {
    int my_intoken = 1, my_outtoken = 1, go_on;
    int all_clear = TRUE;

    while (TRUE) {
        int flow_exec_info = COROUT_WAIT;

        /* step 1: wait until ‘‘something’’ happens, e.g: new user io */
        if (all_clear) AVScorout_X_wait (..., &flow_exec_info, ...);
        else           AVScorout_wait();

        /* step 2: get new inputs and parameters, if they have changed */
        if (flow_exec_info == COROUT_WAIT)
            AVScorout_input (..., &my_intoken, ..., &go_on, ...);
        }

        /* step 3: determine whether coroutine can go ahead */
        all_clear =
            (   (my_intoken == my_outtoken) /* sync. handshake completed */
             || (my_intoken == 0)           /* no synchronization */
             || (go_on)                     /* override synchronization */
           );

        if (all_clear) {
            /* step 4: do the work, e.g: get, process all new user io */
            ...

            /* step 5: if no new interaction, generate new token */
            if (new_user_io) {
                my_outtoken = my_intoken+1;
                all_clear = FALSE;
            }

            /* step 6: send appropriate outputs, mark rest ‘unchanged’  */
            AVScorout_output (..., my_outtoken, ...);
        }
        else {
            /* step 7: save all other changed inputs or parameters */
            ...

            /* step 8: send appropriate outputs, mark outtoken ‘unchanged’  */
            AVScorout_output (..., my_outtoken, ...);
        }
    } /* infinite loop */
}

/*
/* Skeleton module to forward tokens
*/
module_description () {
    AVScreate_input_port (‘‘ext_intoken’’, ‘‘integer’’, OPTIONAL);
    AVScreate_output_port (‘‘ext_outtoken’’, ‘‘integer’’);
}
module_compute (..., int ext_intoken, ..., int ext_outtoken, ... ) {
    /* do the work */
    ...

    ext_outtoken = ext_intoken;
}
```

Figure 3: Skeleton coroutine and module to process and forward token flow

SEVERAL COROUTINES AND ONE MODULE

Many applications need to be able to use more than one coroutine. For example, several “Display Data” coroutines can be used to view several data sets in different windows side-by-side. Each coroutine displays and monitors events in one window, forwarding user interaction into the data flow network. Linked cursors between any number of windows can then be established by connecting the outputs of several coroutines with a Multiplexer module. The Multiplexer mixes user interaction from either window and forwards it to other modules downstream, such as a Data Probe. Such cursor linking, combined with a window migration capability by which users can allocate their windows on any local or remote X-display, is the gateway to many powerful tele-collaborative data exploration applications [4, 5].

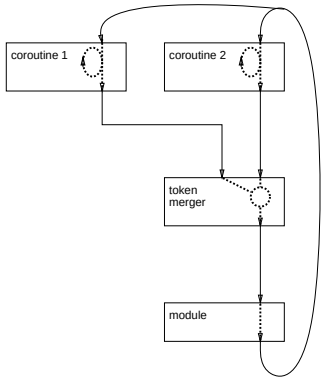

Figure 4: Control flow between two coroutines and one module

In principle, a new, single coroutine could be created to maintain more than one window. Yet, specific window monitoring policies would have to be built into the module. We prefer to separate the basic window monitoring task from higher-level interaction interpretation routines, such as the Data Probe and Multiplexer. This separation allows us to reuse and reconfigure our basic tools much more flexibly, while exploiting the process switching capabilities of UNIX™ to monitor several event loops in parallel.

Figure 4 shows a token distribution pattern with which developers can gain real-time control over the interplay between the coroutines and modules such as cursor linking. When a user interacts with a window the respective monitoring coroutine sends a new token through the token merger to modules downstream like the Data Probe. While one or more modules downstream process the event, the coroutine is blocked, waiting for the token to come back. When a module is finished and returns the token, the coroutine restarts. The second coroutine is unaffected by the token flow due to different token identifiers. Interactions with the second window will cause a token to flow along the second token loop. Conflicting token identifiers from different coroutines can be avoided by folding unique coroutine identifiers into the token identifiers, using a prime number encoding scheme or representing identifiers as strings.

Race conditions can potentially disrupt the proper token flow in this network. If a user interacts with one window and then immediately afterwards with another window – or if the windows exist on two different screens and two tele-collaborating users interact with them simultaneously – tokens from both coroutines can travel through the network at nearly the same time. The Token Merger or the Data Probe modules may not be able to finish responding to the first token before the second arrives. AVS then aborts and restarts the modules, and the first token drops out of the loop. As a result, the respective coroutine cannot complete its handshake and thus waits indefinitely. Such problems can be overcome by using appropriate timeout conditions for maximal waiting periods of coroutines. Another solution is to replace the Token Merging module with a Token Merging coroutine which receives incoming new tokens at its own pace without being restarted by the flow executive. It can buffer all incoming tokens and forward them to modules downstream at the appropriate rate. This scheme will also allow users to assign different priorities to token flow along different paths, thus imposing different floor control schemes on tele-collaborative interactions.

TWO COROUTINES (MUTUAL EXCLUSION)

In many animation or semi-automatic data analysis applications, developers need to be able to arrange mutually exclusive control schemes between several coroutines. In such applications, coroutines are needed 1) to automatically generate new simulation data, and 2) to display the data and have users interact with it. During the simulation process, the simulating coroutine continuously supplies the displaying coroutine with new data. For maximum efficiency, it has to synchronize its simulation steps with the refresh rate of the display routine, waiting for the display routine to finish before it performs the next simulation step. Vice versa, the display routine needs to synchronize with the simulation routine while the user interacts with the data, waiting for the simulation routine to respond to the user input before accepting further user interaction. Figure 5 presents the appropriate token flow needed to warrant efficient interplay between two mutually exclusive coroutines. The figure shows two crossing loops, one being initiated by each coroutine and flowing through the respective other coroutine.

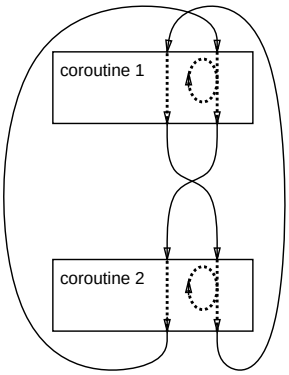

Figure 5: Control flow between two coroutines

We have applied this mutually exclusive handshaking configuration to a semi-automatic, physically-based image segmentation system, “Snakes”, in several biomedical applications [2]. In the Snakes system, users can interactively outline a contour on an image. The snakes simulation iteratively smoothes the contour and adapts it to edges in the images. Users can also interactively alter the snake position by pulling it with the mouse. Snakes were initially implemented as a stand-alone, monolythic program [3]. We decided to break it up into a collection of AVS modules and coroutines so that the code would become easier to adapt to the pre- and post-processing needs in different applications and to telemedicine scenarios [6]. We needed the mutually exclusive handshaking arrangement between a Snakes simulation coroutine and a Display Data coroutine to achieve interactive speeds that were comparable with the stand-alone system. The result is a cascade of mutually exclusive synchronized independent process – including also the X-server – so that user interaction ripples without delay from the windowing system to the Snakes simulation and back to the redisplayed window, avoiding any unnecessary and uncoordinated intermediate redisplays of invalidated data.

CONCLUSIONS

Visual programming interfaces and data flow are great advances in scientific visualization environments towards providing users with modularized, and more customizable visualization tools. Yet, they come at a price: It is hard to achieve the interactive speeds comparable to special-purpose, monolithic programs. To improve the efficiency of interplaying coroutines and modules, we have added token flow to the AVS data flow scheme. Here is a list of conclusions we have drawn from our experiments:

- Token flow is a very powerful process synchronization tool because it lets developers specify

interactively what execution constraints exist between any number of modules and coroutines, i.e., the visualization framework does not have to impose any particular policy.

- It is very easy to add optional token forwarding to modules. All modules should provide this service, in case a developer needs to get this feedback from the module in a particular application.
- Token forwarding is equally easy to add to coroutines, and should be provided, in case coroutines need to coordinate a mutually excluclusive execution pattern.
- Adding proper token generation and token evaluation to a coroutine is rather hard because the coroutine code has to be modified in several different places. Particular care has to be taken to ensure that no parameter or input changes are skipped while the routine idles if input and output tokens do not match. Furthermore, it is easy to create infinite loops between mutually exclusive coroutines, resulting from void handshaking back and forth: output ports (outtokens) need to be marked “unchanged” whenever they do not bear new information. Finally, shared memory can cause headaches because several coroutines and modules will actually share the memory locations of intokens and outtokens. Token values can mysteriously change in the middle of an execution, due to updates from another, concurrent coroutine. It is thus critical to copy intoken values into local variables immediately after they have been obtained, and to update outtokens only when they are sent out.
- We need higher-level tools to manage control flow. Some examples, such as a Token Merger and a Multiplexer were mentioned in this paper. Tool libraries should also include looping and conditional constructs. Many such concepts and protocols have been discussed in the data flow research community.
- Due to the added token flow, even simple AVS networks are becoming very hard to understand. Users and developers would benefit greatly, if the AVS visual programming interface could provide facilities to specify and view several, semantically distinct, subgraphs of the entire network.

In conclusion, current visual programming tools in AVS are not yet quite adequate for real-time, highly interactive data exploration applications. The system can be tuned for such purposes by imposing a token control scheme on the AVS data flow system. With hard work, efficient execution patterns can be created. We look forward to the future development of higher-level scheduling tools within the AVS system, allowing users to achieve good performance more easily.

ACKNOWLEDGMENTS

I am grateful for the many discussions with the Visualization Group at the Cambridge Research Lab (I. Carlbom, W. Hsu, R. Szeliski, D. Terzopoulos, K. Waters), as well as with R. Nikhil and M. Tuttle about various aspects of improving the interactive speed of my AVS networks. A. Payne's showlog program was invaluable in helping me visualize the timing constraints between mutually exclusive coroutines.

References

[1] W. Bethel. AVS tricks: Coroutines and animation. *AVS Network News*, 2(3):15–16, Summer 1993.

[2] I. Carlbom, G. Klinker, and D. Terzopoulos. Soft tissue segmentation for medical images using interactive deformable contours. In *Submitted to Visualization in Biomedical Computing 1994*, Mayo Foundation, Rochester, MN, Oct. 4-7 1994. VBC.

[3] I. Carlbom, D. Terzopoulos, and K.M. Harris. Reconstructing and visualizing models of neuronal dendrites. In *Proc. of CG International '91: Visualization of Physical Phenomena, Boston, MA*, Tokyo, Japan, June 24-28 1991. Springer-Verlag.

[4] G.J. Klinker. An environment for telecollaborative data exploration. In *Proc. Visualization '93*, San Jose, CA, Oct 1993. IEEE Computer Society Press.

[5] G.J. Klinker. Interactive data exploration and telecollaboration in biomedicine using AVS. In *Proc. of the 2nd Int. AVS User Group Conference*, Walt Disney World Dolphin, FL, May 24-26 1993.

[6] G.J. Klinker, I. Carlbom, W. Hsu, and D. Terzopoulos. Scientific data exploration meets telecollaboration. In *Submitted to the 2. ACM International Conference on Multimedia*, San Francisco, CA, Oct. 15-20 1994. ACM Press.