A NOVEL APPROACH TO TOXICOLOGY SIMULATION BASED ON AUTONOMOUS OBJECTS

Munehiro Fukuda, Katherine L. Morse, Lubomir Bic, Michael Dillencourt
Department of Information & Computer Science
Edward Lee, Daniel Menzel
Department of Community & Environmental Medicine
University of California, Irvine
Irvine, CA 92717
email: mfukuda@ics.uci.edu

KEYWORDS: toxicodynamic models, pharmacokinetic models, autonomous objects, distributed simulation, cooperative work.

ABSTRACT

Toxicology simulation is a computationally intensive application which can significantly benefit from tools that not only improve performance, but also allow medical specialists to interact with the simulation and each other in ways which are flexible and intuitive. MESSENGERS is a new computing and coordination paradigm for distributed systems, based on the concept of autonomous messages that carry their own behavior and that propagate autonomously through the underlying computational network. MESSENGERS is inherently distributed, and provides flexibility and intuitiveness for medical rather than computer specialists. We show how MESSENGERS can support such distributed, cooperative, interactive toxicology simulations.

1 MOTIVATION

Historically, decisions on toxic and hazardous materials have been made on a qualitative basis. Because of the enormous costs involved and the potentially devastating consequences resulting from error, the qualitative approach is inadequate. Several approaches have been developed which extrapolate from experimental animal studies or accidental human exposure to toxins. While the initial measurements are quantitative, the extrapolations themselves are done qualitatively and, thus, lack reliability. As a consequence, a number of mathematical approaches to extrapolation of dose have been developed [AK94].

While such mathematical modeling offers many advantages, its use has been restricted to a small number of toxicologists because it is difficult and time consuming, both in terms of developing the model itself and working with the resulting model to produce results. Our resolution of these issues is based on the following insight. There is inherent parallelism and autonomy in the modelled biology. In the body, all organs operate continuously and simultaneously. The flow and interaction between organs is minimal compared to the amount of processing individual organs perform as a whole. Computationally and intuitively this is a more tractable approach than previous tightly-coupled models. On a distributed system, we can map organs to nodes (or subdivide organs into finer grids and map the grid cells to nodes). The inherent scalability of the approach makes it good for performing larger, more complex simulations by adding more nodes to support more organs or finer grids. Most importantly, the user can now focus individually on the function of each organ, providing more or less detail as appropriate for the goals of the simulation.

In the next section we describe the MESSENGERS paradigm and language. We then show how MESSENGERS can be applied to toxicology simulation, including how to support cooperative environments and dynamic manipulation of the model.

2 THE MESSENGERS PARADIGM

At the most abstract level, a distributed computing system consists of a collection of nodes communicating with each other via messages. The computations at each node are instruction streams executing asynchronously and operating on local non-shared data. Message passing is used to achieve collaboration among the different asynchronous streams of control toward a common goal.

Within this generic computational framework, different programming paradigms have been developed to describe how data and control are to be distributed and how information can be communicated between the different streams. In most cases, a conventional programming language is extended with primitives to provide for explicit communication and synchronization [BST89]. This includes various forms of send/receive, broadcast, and multicast primitives, as well as some higher-level constructs, which are usually constructed on top of the lower-level primitives. What is characteristic of these message-based paradigms is that the system's control or "intelligence" is embodied in the communicating node programs, while messages carry only simple pieces of information in the form of passive data.

MESSENGERS is a novel paradigm for distributed computing which departs from the above conventional view by elevating messages to higher-level entities that embody some degree of autonomy or "intelligence." A message can be viewed as an object itself, which has its own identity and which can decide at runtime where it wishes to propagate next and what tasks it is to perform there. MESSENGERS exhibits two novel characteristics which differentiate it from earlier message-based paradigms:

- *autonomy* – the ability of each Messenger[*] to navigate through the network, to possibly replicate itself, and to decide what actions to take at each destination
- *coordination* – the ability of each Messenger to invoke and control the execution of conventional sequential functions residing on various nodes throughout the network

2.1 The Language of MESSENGERS

Over a distributed computing system, Messengers dynamically construct an application-dependent logical network. Multiple logical nodes can be mapped onto the same computer. Each link of the logical network has a name and several (optional) weights, which Messengers use for determining which links to route themselves along. Each logical node has a name and provides a memory space commonly accessed by all Messengers that gather on the node. This memory space, the Node Variable Area, functions as both a node-unique database and as an inter-Messengers communication channel.

Each Messenger includes three types of variables: messenger, node, and environmental variables. Messenger variables are local to and carried by the Messenger as it propagates through the logical network. Node variables are node-resident and are mapped to the Node Variable Area where the Messenger is currently running. There they are shared among Messengers running on the same node. Environmental variables provide information such as the current node name, and the name and weights of the last link traversed. All types of variables are accessible at any point in time. Every Messenger program has the following form:

$$\text{node variables; messenger variables; S1; }\overset{\downarrow}{\text{S2}}\text{; ... Sn;}$$

[*]We use capitalized lower-case when referring to the individual autonomous objects and upper-case for the entire system.

The node variable declaration allocates the node variables in the Node Variable Area whenever the Messenger arrives at a node which has not yet been visited by a Messenger in the current application. Since the node variables are shared by all Messengers in the application, they must be allocated only once by the first Messenger to arrive at the node. In addition to the usual types of variables, a messenger variable may be a character string containing the name of a precompiled functions to be carried by the Messenger for the purpose of remote evaluation. Environmental variables are predefined and referred to without explicit declaration. The declaration format is similar to that of the C Language and provides a variety of data types excluding pointer, union, and unsigned.

The remainder of a Messenger program consists of statements, denoted by Si. The arrow is used to indicate the current statement, i.e., the one to be interpreted next. This corresponds to a program counter in a conventional language and is saved in a status block of the Messenger. Since the Messenger migrates between different nodes, the status block including this must also be carried. Each Si is one of three types of statements: computation, navigation or function invocation. Computation statements are either assignment or control constructs as in C. Navigation and function invocation statements are MESSENGERS-unique. The following is a complete list of MESSENGER statement types:

Computation: A Messenger may perform arbitrary computations using assignment and control statements from the C programming language. These statements are all interpreted by the current node.

1. *assignment*: The Messenger may read and update node variables and messenger variables, and read network variables. Arbitrary expressions are permissible in the assignment and include all of the common arithmetic and logic operators provided in C.
2. *control statement*: The Messenger may perform all the common control statements supported in C, such as if-then-else, while, do-while and break.

Navigation: A Messenger may create new logical links and nodes, change or delete existing ones, and move arbitrarily though the network by following links or jumping to specific nodes.

1. *create(node, link, weight, physical)*: Create a *link* along which the Messenger moves, associate *weight* with this *link*, and create *node* on the *physical* node which the Messenger reaches. All parameters are optional. If omitted, the new node/link has no explicit name or weight and is created on a physical node chosen automatically by the system. Hence the last parameter allows the user to explicitly control the logical-to-physical node mapping.

2. *hop(node, link, weight, physical)*: Cause the Messenger to be forwarded to *node* on *physical* node along *link* if it has the associated *weight*. As with *create*, all parameters are optional, which offers great flexibility in specifying the Messenger's next node destinations. If there are multiple destinations, a copy of the Messenger is propagated to each destination. If a non-existent destination is specified, the Messenger is terminated. In addition, the parameters support "wild card" matching, which results in a powerful multicast mechanism for each Messenger. For instance, hop(node=*; physical='laguna') sends a copy of the Messenger to all nodes on the physical node named 'laguna', regardless of any logical links. The navigation hop(node=a) propagates the Messenger along all logical links emanating from the current node to all neighboring nodes named "a".

3. *delete(node, link, weight, physical)*: Forward the Messenger as with *hop*, but also delete the links traversed by each copy of the Messenger. In addition, if this action erases all links from the departing node, the node is also deleted.

Function Invocation: This allows the Messenger to load and execute ordinary C functions, precompiled and residing on the current physical node.

1. *exec(filename, arguments, [nowait])*: Invoke a precompiled function specified by *filename* as a separate concurrent process and pass *arguments* to it. If *nowait* is specified, the function may continue operating even after the Messenger has left the node (or has terminated).

2. *func(filename, function_name, in_arg, out_arg)*: Invoke a precompiled function specified by *filename* and *function_name* as a part of the MESSENGERS interpreter, wait for its completion, and return the results back to the invoking Messenger program. The *in_arg* and *out_arg* are arguments passed to and returned from the function respectively. The invoked function also receives pointers to the node variable area and the memory space holding the messenger variables and thus can manipulate them directly. The loading of the function is triggered dynamically when it is invoked for the first time.

2.2 The MESSENGERS Interpreter

The MESSENGERS system is implemented as a daemon running a language interpreter at each physical node. The daemon exchanges Messengers with other daemons and multiplexes logical nodes mapped to its physical node.

For each new Messenger, the interpreter continues processing its statements until the interpreter encounters a statement that is navigational or functional, i.e., one that causes the Messenger to be removed from the current node or to be suspended when a precompiled function is invoked as another process.

Figure 1 illustrates these basic principles graphically. It shows two network nodes, each running the MESSENGERS interpreter as one of its applications. A Messenger is received by the first node (step 1) where the current statement, c_i, is assumed to be navigational, e.g. hop. The Messenger is forwarded to the next node (step 2). The next statement, c_{i+1}, is assumed to be a func() statement which causes the invocation of a node-resident program f() (step 3). Upon returning from the function call, the interpreter carries out the next statement, c_{i+2}, which we assume to be sent to all neighbors of the current node (step 4), with the current statement pointer at statement c_{i+3}. Note that steps 3 and 4 are sequential since the propagation awaits the completion of the function. If, however, an exec() command with a *nowait* parameter were used instead of "func", the two steps would proceed concurrently.

2.3 Simulation Library

A MESSENGERS interpreter can incorporate a set of precompiled functions as a user-defined library upon its initialization. We provide a simulation library that implements virtual time control, entity control, and event scheduling. By invoking these functions, Messengers can act as simulation entities, executing computations according to virtual time.

Virtual Time Control: The virtual time control functions, *gvt_start()* and *gvt_stop()* start and stop an optimistic simulation, in which each node continues to process events without synchronization until it has a virtual time inconsistency with its neighbors. When a node receives a Messenger whose virtual time is earlier than the current node's, it locally rolls back the simulation to the appropriate virtual time.

Entity Control: The entity control functions, *duplicate()*, *inject()*, and *kill()* duplicate the calling Messenger, load and start a new Messenger, and terminate the specified Messenger respectively. The generated Messenger's identifier is returned to the calling Messenger.

Event Scheduling: When GVT is active, Messengers can schedule certain events to occur relative to the GVT. The events are processed independently in each physical node. The event scheduling functions may be called from any user-defined function, *f*, invoked using the *func()* statement. They schedule future events for either the calling Messenger or some other Messenger. When a Messenger schedules any event for itself, it is automatically suspended at the end of the invoking function *f*. Events are processed by the MESSENGERS interpreter in the order of their GVT time stamps. Each event is tied to a particular Messenger and it causes the reinvocation of the same function *f* that caused the Messenger's suspension. Below we will refer to this function as the latest function:

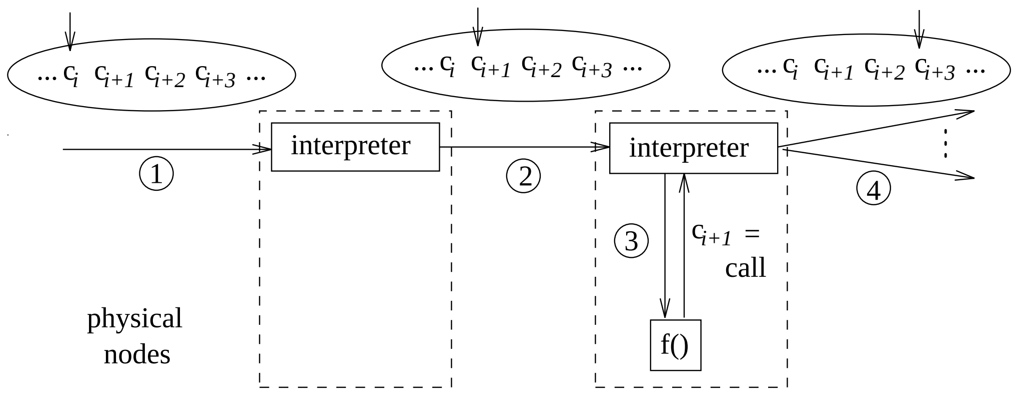

Figure 1: Messenger Invokes Node-Resident Program f()

1. *sched_self(time):* Schedule the re-invocation of the latest function of the calling Messenger at GVT= *time*.

2. *sched_other(mid, time):* Schedule the re-invocation of the latest function of the Messenger identified by *mid* at GVT= *time*.

3. *sched_at_node_var(node_variable, condition, constant, time):* Schedule the re-invocation of the latest function of the calling Messenger when *node_variable* and *constant* satisfy *condition*, for example, when for some node variable $N > 0$. *time* gives the maximum waiting time for this to occur.

4. *sched_at_slope(messenger_variable, constant, time):* Schedule the re-invocation of the latest function of the calling Messenger when the rate of change of the *messenger_variable* exceeds *constant*. *time* gives the maximum waiting time for this to occur.

5. *sched_at_threshold(messenger_variable, condition, constant, time):* Schedule the re-invocation of the latest function of the calling Messenger when *messenger_variable* and *constant* satisfy *condition*. *time* gives the maximum waiting time for this to occur.

6. *cancel(mid):* Cancel all future events associated with the Messenger identified by *mid*.

3 APPLYING MESSENGERS TO TOXICOLOGY SIMULATION

Most toxicology models are based on dosimetry: the concept that chemical toxins are distributed into discrete compartments within the body. The first organized attempt at such modeling appeared in 1984[MM84]. One approach to dosimetry modeling of inhaled, ingested or absorbed chemicals has been that of Andersen and his coworkers [AK94, RA84]. This approach has become very popular as many individual chemicals have been modeled, and it has become known as physiologically based pharmacokinetic (PB-PK) modeling. PB-PK models became popular because they use the anatomic and physiological relationships of individual organs in the body to develop a multicompartmental mathematical model.

The PB-PK model can be generalized into a physiologically based component and a chemically specific component. At present PB-PK models are developed as one integrated model for one chemical. The model is written specifically for the metabolism of a given chemical. Certain organs may be lumped together in order to simplify the solution of the model. No single standard model that could be used to develop chemical specific models now exists.

Another type of modelling, referred to as "toxicodynamic modeling," is based on statistical models of cell dynamics. Once the dose is defined at the site of action of the chemical, the relationship between the exposure concentration, route of exposure and time of exposure to the biological effect of the toxin can be developed. Toxicodynamic modelling is more complex and requires additional expertise in statistics. It may be possible to define a standard dynamic model for specific organs, making possible even more reliable and realistic estimates of human risks.

Figure 2 depicts a schematic description of a typical PB-PK model of inhaled styrene [KAB93], as it would be represented in MESSENGERS. The amount of styrene and its metabolites from a given exposure to airborne styrene can be simulated in each compartment.

PB-PK and toxicodynamic simulation models (e.g. Figure 2), can be implemented as networks of nodes communicating with one another via messages. During the model construction process, the user selects the equations to govern the behavior of each node. The programs implementing each equation are invoked by Messengers circulating though the system along the directed arcs. For example, the lung node, having computed the

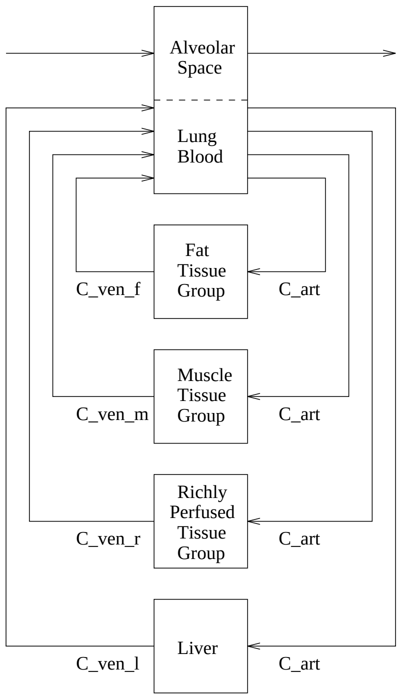

Figure 2: PB-PK Model of Styrene Inhalation

arterial concentration C_art, sends a Messenger to each of the other organ nodes in parallel. This Messenger carries the new value and a sequence of function names to be invoked in each receiving node. When a node computes the new venous concentration C_vf, the Messengers collaborate to collect the results from all organ nodes and sum the values to obtain the total venous concentration. In this manner the Messengers circulate through the model in much the same way as the blood carrying the toxin would circulate through the body. This circulation, which corresponds to the integration process in a conventional simulation model, continues until one of the following conditions occur:

- the total simulation time set by the user has been reached
- the user suspends the simulation by sending a signal to one of the nodes (e.g. the lung)
- a breakpoint, set by the user, has been reached; breakpoints have two possible forms:
  - the simulation pauses after a given interval of simulated time has elapsed
  - the simulation pauses when a certain condition has been satisfied in some node, for example, the concentration in an organ has reached a certain value.

3.1 Dynamic Model Manipulation

Dynamic model manipulation encompasses the ability to change both the definition and the state of a simulation experiment while it is in progress. This is referred to as "computational steering." The next set of actions or parameters is determined based on intermediate results already observed from the current simulation. With most existing simulation and modeling tools, the entire experiment must be set up *a priori* and only the final output may be used as feedback for the next experiment. To be most effective, the tool should support the ability to make incremental progress, allowing the user to backtrack to an earlier point and continue the experiment in a different direction.

The MESSENGERS paradigm offers a strong foundation for such flexibility in that each message arriving at a node carries not only intermediate value but also prescribes the sequence of functions to be invoked in that particular node. When the MESSENGERS circulation pauses (for example as a result of a breakpoint), the user may modify the composition of any Messenger. For example, the user could decide that a given organ should compute its venous concentration using not on ly the kinetic flow concentration and amount but should also consider metabolism. This could be achieved by modifying the Messenger flowing to that organ at each time to invoke the corresponding metabolism equation as part of the concentration computation in that organ. In current simulation control language approaches, the decision to add a metabolic step to an organ requires a major revision of the entire model.

3.2 Cooperative Environments

Thus far we have only addressed the needs of an individual researcher working alone. However, we also recognize the need for multiple researchers to interact with a given simulation concurrently. This ability facilitates collaboration, allowing researchers from different institutions to participate actively in the same experiment without having to come together to the same physical facility.

The MESSENGERS paradigm is open-ended in that every node is capable of receiving and interpreting MESSENGERS programs arriving from any destination. Hence we can control the simulation from any of the participating workstations. This naturally extends to wider distributions, on which different users are located at arbitrary geographical locations. We envision MESSENGERS as providing the foundation for an Internet-based cooperative environment. The environment will support not only distributed simulation, but computer teleconferencing. With the addition of a GUI which frees users from tedious programming details and allows them to focus on the application, expert knowledge and training can be pooled to enhance the understanding of the basic concepts of mathematical modeling of toxicological events and to train new modelers.

4 CONCLUSIONS

In this paper, we have demonstrated the need for quantitative toxicology simulations which are accessible to medical specialists. We have also provided an overview of the MESSENGERS paradigm and language, and shown how MESSENGERS can support such simulations by providing an intuitive biological model in which the organs are simulated in parallel.

As a preliminary effort to demonstrate the feasibility of using this paradigm in this context, we have implemented a prototype model of the inhalation of styrene. The model is based on that of Ramsey and Andersen[RA84]. We constructed the model using the same anatomical compartments as Ramsey and Andersen, but instead of using a simulation control language we used the autonomous objects paradigm. We compared the computed values with those of Ramsey and Andersen and found good agreement.

Based on our experience using this paradigm we designed MESSENGERS with the following improvements: sharing of node variables between Messengers and running programs written in other languages, efficient implementation of functions to support synchronization primitives, virtual time, higher level language specification, efficient implementation of the interpreter as a single daemon, efficient parameter evaluation and passing. An initial implementation of MESSENGERS [Bic95] [Fuk95] for networks of Sun workstations is now completed and is being used to develop a more efficient versio of the styrene inhalation model and other simulation models. We are also developing a graphics inteface to alleviate the burden of conventional programming.

References

[AK94] M.E. Andersen and K. Krishnan. Physiologically based pharmacokinetics and cancer risk assessment. *Environmental Health Perspectives*, 102 Suppl 1:103–108, 1994.

[BCS94] P.M. Borst, M. Corbin, and P. Sapaty. WAVE processing of networks and distributed simulation. In *Third IEEE International Symposium on High Performance Distributed Conference*, San Francisco, CA, August 1994.

[Bic95] L.F. Bic. Messengers: Distributed computing using autonomous objects. Technical Report 95-19, University of California, Irvine, Irvine, CA, May 1995.

[BST89] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Surveys*, 21(3), March 1989.

[Fuk95] Munehiro Fukuda. *MESSENGERS: Language Specification and System Interface*. Department of Information and Computer Science, University of California, Irvine, Irvine, CA, msgr-01 edition, July 1995.

[KAB93] L.B. Kedderis, M.E. Andersen, and L.S. Birnbaum. Effect of dose, time, and pretreatment on the billiary excretion and tissue distribution of 2,3,7,8-tetrachlorodibenzo-p-dioxin in the rat. *Fundamental and Applied Toxicology*, 21:405–411, 1993.

[MM84] F.J. Miller and D.B. Menzel, editors. *Fundamentals of Extrapolation Modeling of Inhaled Toxicants: Ozone and Nitrogen Dioxide*. Hemisphere Publishing Corp., Washington, D.C., 1984.

[RA84] J.C. Ramsey and M.E. Andeersen. A physiologically based descriptionof the inhalation pharmacokinetics of styrene in rats and humans. *Toxicology and Applied Pharmacology*, 73:159–175, 1984.

BIOGRAPHIES

The MESSENGERS project is a collaboration between Drs. Bic and Dillencourt, and students in UCI's ICS department: Munehiro Fukuda, Bozhena Bidyuk, Jason Cahill, Susan Mabry, Fehmina Merchant, Katherine L. Morse, and Alex Thornton. The toxicology simulation is a collaboration between the MESSENGERS project, and Dr. Menzel of Community & Environment Medicine and his student Edward Lee. Additional information about MESSENGERS can be found on the WWW at http://www.ics.uci.edu/ bic/messengers/messengers.html. Copies of publications [Bic95, Fuk95] are also available via this web page.