An Overview of Hermes

Amr El-Kadi
Shmuel Rotenstreich

Department of Electrical Engineering and Computer Science
The George Washington University
Washington DC 20052

With the growing popularity of distributed systems, the implementation of these systems will be inappropriate with conventional programming languages that use system calls. Languages designed for implementing distributed systems should be able to deal with the basic issues that differentiate distributed programming from sequential programming [1], these are as follows:

1. The use of multiple processors that do not share physical memory (whether these are *closely or loosely coupled*),
2. The cooperation among processors by sending messages over the *communication network*,
3. The ability to cope and continue functioning in the existence of *partial failures*.

Hermes is a very high level language aimed at the development of large distributed systems. It is an ongoing research project being done at IBM T. J. Watson research center. It has evolved from previous work done with NIL (Network Implementation Language) which was used for the implementation of IBM's SNA communications software. Hermes is *a secure language* according to Hoare's requirement in his 1981 Turing Award lecture in which he criticized Ada's design for not being secure [3]. In a secure language according to Hoare's requirement, a bug in module X should show up as bad output from module X, neither as a system crash, nor as a bad output in a correct module Y [2]. Security in Hermes is the result of the invention called typestate checking [4].

In section 1, a brief description of *typestate* checking will be given. In section 2, a description of how parallelism is achieved in Hermes will be explained. In section 3, communication mechanisms supported by Hermes will be discussed. And finally, in section 4, Hermes fault tolerance will be overviewed. In these sections the intention is to demonstrate the main features of Hermes, and not to cover comprehensively all of Hermes constructs. A comprehensive presentation of Hermes can be found in [2].

1 Typestate Checking

Hermes does much more *compile-time* checking than languages such as Ada or Concurrent C. The additional compile-time checking is the result of applying typestate checking- a generalization of *dataflow analysis* [2]. The advantage of typestate checking becomes clear when we discuss what causes programs to violate security as defined by Hoare [3]. An *erroneous execution* is the result of unchecked violations of language rules [2][6]. Erroneous executions mainly have been attributed to:

1. Accessing data with undefined values,
2. Accessing data of the wrong type.

This paper was originally published as *Technical Report # GWU-IIST-91-24, December 1991, Department of Electrical Engineering and Computer Science, The George Washington University, Washington DC 20052*

Conventional type checking deals with catching errors of accessing data of the wrong type. Typestate checking deals with catching errors of accessing data with undefined values. Hermes uses typestate checking at compile-time to keep track where a variable is uninitialized and signaling any attempts to read it as an error, where a variant is in the wrong case and signaling any attempts to read/write it as an error, and where a program value is unchecked (compiled successfully) and signaling any attempts to instantiate it as an error.

To give a simple example, consider the following three statements written in *pseudocode* for simplicity [1]:

1. X,Y: Integer;
2. if condition then X := 4; end if;
3. Y := X + 3;

In any conventional programming language such as Ada [6], Concurrent C [7], or Modula-2 [8], this code segment would compile correctly, and in some cases it will either cause the program to crash, or to give wrong output. This will be caused (if it occurs) depending on the value of the condition in statement 2. If the condition evaluates to true, then variable X will be initialized and statement 3 will be meaningful. But in case the condition in statement 2 evaluates to false, then variable X will be uninitialized and the execution of statement 3 will result in assigning Y garbage. This example shows the vulnerability of conventional programming languages to such errors that stem from accessing uninitialized variables. In Hermes, this code segment would have been rejected at compile-time signaling a *typestate error*.

A typestate is represented as a *set of attributes* for variables [2]. Examples are whether a variable is initialized or not. If a variable A is initialized, then the attribute **init**(A) will be present in the typestate, the absence of this attribute indicates automatically that the variable is uninitialized. At compile-time, each variable at any source statement is known to be initialized or not. Sometimes *coercion* is used to resolve uncertainties, for example making an initialized variable uninitialized if the rules indicate so. A typestate with a strict subset of attributes of another typestate is said to have *lower typestate*. To ensure security, coercion always converts from *higher* typestate to *lower* typestate.

In Hermes each operation has *precondition rules* which indicate the set of required and forbidden attributes, and *postcondition rules* which indicate how the typestate changes after the application of the operation, that is the set of added and dropped attributes from the precondition rules [2]. Since any operation can result in an *exception*, then a different set of postcondition rules may also be specified for exceptions. When an exception is raised, then the control jumps to the nearest matching *exception handler clause* in an enclosing block in the process in which the exception is raised. This makes it impossible to ignore exceptions.

In Hermes, a program is said to be *correct* if starting with an initial typestate in which the only initialized variable is the *initialization port* (initialization ports will be discussed in the section on parallelism), it is possible to mark each point in the source program with a typestate and the typestate at each point meets the precondition rules of the operation, and the typestate after the execution of the operation is derived using the postcondition rules [2]. The program will be rejected if an invalid typestate results as a side effect of applying postcondition rules. The set of *valid typestates* are defined in [2].

The *typestate analysis algorithm* is applied at each statement starting with the existing typestate. Each application propagates a new typestate to successor statements. The following steps are repeated until all program statements have typestates and no loop needs to be re-analyzed [2][4]:

1. Determine the precondition for the operation using the operation specifications specified in the language reference manual if the operation is predefined or using the specifications supplied by the user.
2. If any required attribute is missing, then indicate an error.

3. If any attribute is present which is forbidden, then try using coercion to drop any such attribute. If the result of applying coercion drops an attribute that is required by the preconditions, then indicate an error. Otherwise we are successful so far.
4. Generate an error if the operation modifies any variable that is specified as a constant.
5. Determine the set of all possible normal and exception outcomes of the operation.
6. Determine the postcondition for each outcome, and use it to transform the precondition to typestates (one for each outcome).
7. Check that the typestates that resulted after step 6 are valid.
8. When each typestate is computed and checked in steps 6 and 7, then propagate them to the appropriate destination. If the destination already has a typestate, then compute the meet (the intersection) of both typestates. Coercion may be used if needed.

For each loop construct in Hermes, typestate checking is done with a single *backward branch* from the end of the loop to the beginning. If the typestate at the end of the loop is greater than or equal to the typestate at the beginning, then nothing is done except coercion is used if necessary. Otherwise, the meet of both typestates is taken, thus lowering the typestate at the beginning of the loop and then the statements of the loop are re-analyzed. It was found that in practice, no loop will need to be re-analyzed more than twice [2][4].

2 Parallelism

Hermes is a *process-oriented* language, that is a Hermes program is a set of processes running in parallel. In Hermes (and in NIL as well), a process is also the unit of modularity which makes Hermes more uniform than languages that have separate mechanisms for expressing parallelism and modularity (e.g. tasks and packages in Ada) [1][2]. As in other languages for programming distributed systems, processes may run in parallel on physically distributed processors, or they may run on one or more processors in *pseudoparallelism*. The distinction between true parallelism and pseudoparallelism is transparent to the programmer as the interaction between processes is the same whether processes are local (running on the same processor) or remote (running on another processor). Hermes takes it even further by having uniformity in interaction between processes whether they belong to the same application or not [2].

In Hermes, there is no data sharing among processes and each process has its own local variables and its own code. Also there are no pointers or aliasing. Processes in Hermes synchronize and cooperate through message passing, as will be explained in section 3.

A process in Hermes is a sequential program with inputs, outputs, and local memory [2]. Processes in Hermes are *instantiated* (created) explicitly using the **create of** operation to a value of type program. The value returned by the **create of** operation is an output port connected to the *initialization port* of the created process (*the child*). The initialization port of the child process is an added flexibility [1]. Through this connection, the creator of the child (*the parent*) can pass parameters to the child to give it connections to the outside world (to other processes) as well as any other data needed by the child. In languages like Ada, in which processes (tasks) do not take parameters when they are created, a special mechanism is needed to send *set-up parameters* to the newly created tasks (e.g. by having dedicated accepts for that purpose) [1][6].

Hermes uses a *capability-based* model for getting services[2], you can access a service that you do not provide by having the *capability* to do so. This capability is in the form of an output port that is connected to an input port which belongs to the server, and which the server designates for receiving requests for services of the kind you want to perform (input ports and output ports will be discussed in section 3). So, the parent of a process is the one who initially gives its child the ability to get services by passing it capabilities to do so, and this also includes the ability to do I/O [2]. Capability-based models have been known to ensure security in operating systems but it was not used widely until recently because capabilities are costly to implement [2]. But since they are *impossible to forge*, ways for implementing them efficiently were established and applied as in the case here in Hermes [2] and in the Amoeba distributed operating system [9][10]. Capabilities in Hermes can be created using the **connect** operation, they can be copied or passed around in messages. Capabilities given to processes can later be revoked by the process providing the service. This feature gives Hermes advantage over *object-oriented* languages in a sense that when you have access to an object in object-oriented languages, you have automatic access to all its operations. This is not the case in Hermes, a server can allow access selectively to any of the services it is providing and it can dynamically revoke access from any client or all clients.

A Process in Hermes *terminates* when it finishes execution (that is when it hits the **end process** statement) or when it has no connections to the outside world. The later case is special, since a process with no connections to the outside world can be safely considered by the implementation to have no desire to continue existing since no useful work can out come from it, so the implementation can *garbage-collect* it. In Hermes, there are no primitives that give processes (even parents) the ability to terminate other processes as in other languages like Ada and Concurrent C.

A *mapping* mechanism maps processes to physical processors [1]. Different mapping strategies are supported by different languages which vary depending on the needs of the application to be implemented. Basically, applications may have two different goals for mapping [1]. First, mapping may be applied with the goal of increasing parallelism to decrease the turnaround time. This is analogous to *load balancing* in distributed operating systems with the distinction that mapping has the goal of minimizing the execution time of a single distributed application, so there is *no competition* among processes here as in the case of load balancing. Second, mapping may be applied with the goal of increasing fault tolerance. In this case processes and/or data may be replicated on different processors to increase reliability.

Mapping can be *transparent* as in Hermes [1][2], in which case the implementation is free to handle it by the compiler or the run-time system. With the later choice (i.e., the run-time system is doing the mapping), Hermes processes may *migrate* between processors to decrease communication time between processes that communicate most frequently for example by running them in pseudoparallelism while weighing this decision against the loss in parallelism and the effect on the desired overall turnaround time. A description of a Hermes implementation that does process migration for load balancing is given in [11]. Another language that allows processes to migrate is Emerald [1]. Its an *object-based* language that allows objects to migrate from one processor to another. In other languages (e.g. Argus and Concurrent C), the distinction between true parallelism and pseudoparallelism is *not transparent* to the programmer. Thus the programmer explicitly decides which process runs on which processor. This complicates the job for the user of the language, while adding the flexibility required by some applications [1][9].

3 Communication and Synchronization

In distributed systems, processes need to *cooperate* to achieve the requirements of the application. This cooperation includes both *communication* to pass data back and forth between processes, as well as *synchronization*. In Hermes, cooperation is achieved by *message passing*. Hermes supports both forms of message passing, messages can be *synchronous* or *asynchronous*. Communication in Hermes takes place on *ports* which are created dynamically at run-time. Two types of ports exist; *input ports* and *output ports*. The value of an output port is a connection to an input port which is a *message queue* in another process (that is the process that is providing the service) [2].

Hermes does not support cooperation among processes through *shared variables*, since in Hermes processes do not share data. Cooperation among processes through shared variables has its advantages and disadvantages when compared to message passing [1]. The advantages include the fact that shared variables are accessible to any process, whereas in message passing, a message transfers information between two specific processes (if broadcasting and multicasting are not used). Also, an assignment to shared data has an immediate effect that is felt by all processes accessing it, whereas in message passing there is a delay involved between sending a message and receiving it. The principle disadvantage of shared data when compared to message passing is that mechanisms should be provided to ensure *mutual exclusion* between processes attempting to update the same data simultaneously. Examples of these mechanisms are *atomic transactions* as in Argus [1][9][14], *tuple space* in Linda [1], *shared logical variables* in PARLOG [1], and the use of *Rendezvous* to ensure mutual exclusion as in Ada [1][6][12][13] and Concurrent C [1][7].

In Hermes a process can create an input port and export connections to it, thus acting as a server. This essentially means that multiple output ports can be connected to a single input port. A process X in Hermes can get connection to an input port (service) P only under one of the following conditions:

1. The process X owns P and it connects to it,
2. P is the initialization port of a process which process X creates,
3. Another process passes process X a binding to P.

A *callmessage* in Hermes contains the parameters sent by the process that performed synchronous communication (using a **call** statement). An input port together with its associated callmessage type is called an *interface* (which defines the typestate of the parameters of which the callmessage consists). An interface may also define an *exception* and the associated typestate of the parameters of the callmessage when such an exception occurs.

Input ports are created using the **new** operation. This operation initializes the input port variable to an empty queue which has no connections. Connections of input ports can be done using the **connect** statement. Hermes also provides an operation to inspect whether an input port has any outstanding callmessages queued on it or not. This can only be done by the process owning the input port using the **empty of** operation which returns true if the queue associated with the input port being inspected is empty, otherwise it returns false.

The **receive** statement dequeues a message from the queue associated with an input port if one is available and it stores it in a destination variable that should have the same type of the callmessage associated with the input port. Otherwise it *blocks* indefinitely until a message is received. If multiple messages exist in the queue at the time of the execution of a receive statement, then the choice of which one to dequeue is done *nondeterministically*, with the following preserved: (1) messages sent by the same process will be received and dequeued in the order by which they were sent, and (2) every message will eventually be received and dequeued provided that the owner of the input port keeps performing **receive** operations on the input port. The exception **Disconnected** will be raised if the input port has not been connected to any output port, and the owner of the input port performs a **receive** from it. It will also be raised if all output ports connected to an input port have been discarded by their owners, and the owner of the input port performs a **receive** from it.

The **return** statement returns a callmessage to its caller and thus allowing it to resume its execution if it was a synchronous call. After its execution, the callmessage is uninitialized and thus any attempt to access it in the process executing the **return** would have resulted in a typestate error at compile-time. A **return** may also return an exception to the caller. The exception will be raised in the caller and not in the process issuing the **return** statement.

Synchronous message passing in Hermes is achieved using the **call** statement [2]. When a **call** statement is issued, the following steps take place: (1) a callmessage is created, (2) the arguments of the **call** are moved into the callmessage, (3) the callmessage is sent to the input port to which the output port is connected, (4) the **call** statement is blocked until the callmessage is returned by the receiver, (5) when the callmessage is returned, the callmessage is broken down into its components which are copied into the corresponding argument variables, and (6) the callmessage is discarded. Arguments of the callmessage may be *in*, *in-out*, or *out* parameters. Their specifications are given in the callmessage definition. A **call** statement may return normally, or with an exception raised in the caller. An exception may be returned by the owner of the input port if a declaration for that exception is specified in the callmessage definition. Two other exceptions might be returned as a result of the **call** statement. The first is returned when the connection between the input port and the output port is broken (for example it may be as a result of the server revoking the capability form the calling process or that the input port does not exist anymore), then any attempt by a process to issue a **call** statement on the output port will result in the exception **Disconnected** be raised in the caller process when the call is made. The second possible exception is the **Discard** exception. This exception is raised in the caller if the callmessage is discarded by the process owning the input port. A callmessage may be discarded by using the coercion **discard** operation. If the callmessage was of type *CMType*, and the message was discarded by the owner of the input port, then the exception *CMType*.**Discarded** will be raised in the caller [2].

Asynchronous message passing may be issued using the **send** statement. The value of the source variable used in the **send** statement is queued on the input port to which the output port is connected. **Send** does not wait for a **return** from the receiver (i.e., it does not block), and the receiver should not perform a **return** since this is a *one-way communication*. Since **send** does not block, it can not receive **Discarded** exceptions, since this exception occurs as a result of the receiver discarding the callmessage using the coercion **discard** and the caller is not waiting for a **return**. A process issuing a **send** can receive a **Disconnected** exception if the connection between the output port and the input port has been broken.

Asynchronous message passing in general may be implemented in two ways: (1) by simulating asynchronous message passing using synchronous message passing, or (2) by having true asynchronous message passing either provided by the underlying operating system or implemented by the run-time system [15]. In the first way, if process A wants to send a message asynchronously to process C, then the message is sent instead to an intermediate process B which returns directly to process A (i.e., process A does not have to wait) after keeping a copy of the message, and sends it to process C and waits on behalf of process A. Process B is called a *buffer process* or an *agent process* [2][7]. The intermediate process may be required to be created by the programmer explicitly thus adding complexity to the programmer's job, which increases the possibility of errors, or it may be provided transparently by the run-time system, which may be inefficient [7]. In the second way, true asynchronous message passing is provided, but a serious problem is associated with this method. Since it may occur that a process A who is sending asynchronous messages to an input port in another process B, and process A executes faster than process B, then messages sent by process A should be buffered on the input port of process B which may eventually result in *buffer overflow*. The problem of buffer overflow may be solved by two methods: (1) by raising an exception in the process attempting to send messages asynchronously, or (2) by blocking the sender until there are available buffers (this is how it is done in Concurrent C) [1][7]. Blocking the sender in case no more buffers are available may result also in the introduction of unexpected deadlocks [1]. In Hermes, an implementation is free to solve the buffer overflow by either of the two methods, but if it chooses the blocking method, it must guarantee that the process will not be blocked indefinitely, and thus it avoids possible deadlocks that might stem from blocking as in other programming languages [15].

It may happen that a server who can receive requests for services on different input ports may wish to keep looping, waiting to receive, dequeue, and serve any of the calls. This is a *nondeterministic wait*, since the server does not know the order in which calls will be issued, or from whom. A language for distributed systems programming must support construct(s) for expressing *nondeterminism*. In languages like CSP [1], Ada [1][6][12][13], Concurrent C [1][7], and Hermes [1][2], the **select** statement is the construct used to express nondeterminism.

In Hermes, the **select** statement is used for expressing nondeterminism as well as for flow control. It has an optional expression, a set of select clauses, and an otherwise clause [2]. Where a select clause consists of a *guard* and a *clause*. A clause is simply a set of statements. The guard can be

- a *boolean guard*,
- an *event guard*, which names an input port and is treated as a boolean guard with 'true' value,
- a conjunction of a boolean guard and an event guard.

Enabled clauses are those whose guards are evaluated to 'true', and *disabled* clauses are those whose guards are evaluated to 'false'. The way the **select** statement works is as follows:

1. All guards are evaluated. If all guards are disabled, then the **otherwise** clause is executed and the **select** statement terminates.
2. If any of the clauses are enabled, and all such clauses have event guards with associated input ports that are not connected, then the **Disconnected** exception is raised.
3. If there is an enabled clause with either no event guards, or with an event guard who has outstanding message(s), then that clause is executed. If more than one such clause exist, then the choice of which one to execute is nondeterministic, and it is not fair (most other languages that use **select** statements for expressing nondeterminism are also unfair [1][6][7]).
4. If some clauses are enabled, but they all have event guards associated with empty input ports, then the **select** statement blocks waiting for calls indefinitely.

4 Fault Tolerance

In distributed systems, the system should continue to function in presence of processor crashes. This can be achieved by the operating system, the run-time system, or the programmer [1]. Several approaches to handle fault tolerance in programming languages have been suggested. Basically they can be classified into three approaches. First, the operating system or the language run-time system can inform the programmer of the existence of failures (by using exceptions for example), and the programmer is responsible for handling such crashes, maybe by creating new processes to replace the crashed ones. For some applications, this is all that is needed [1]. A second approach is to use atomic transactions [14](e.g. Argus [1][9]). Atomic transactions have the property that a transaction either executes fully and successfully, or it has no effect at all. They are *indivisible,* and *reliable*. For some applications such as banking systems which utilizes ATM machines, this is just what is needed. If an ATM machine crashes in the middle of a transaction, the whole transaction has no effect on the customer's account. The third approach, tries to handle faults transparently from the programmer. This is the case in Hermes. The run-time system may employ the *optimistic recovery* technique [1][5] that periodically inserts checkpoints and keeps logging messages on stable storage for recovery in case of crashes. This is done concurrently with the normal computation so as to have the least effect on the normal functionality of the system.

Acknowledgment

We would like to thank Andy Lowry for his helpful comments and suggestions.

References

[1] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), Sep. 1989, pp. 262-322.

[2] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing*. Prentice Hall, 1991.

[3] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2), Feb. 1981, pp. 75-83.

[4] Robert E. Strom, and Shaula Alexander Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1), Jan. 1986, pp. 157-171.

[5] Robert E. Strom, and Shaula Alexander Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3), Aug. 1985, pp. 204-226.

[6] U. S. Department of Defense. Reference Manual for the Ada Programming Language. United States Department of Defense, 1983.

[7] Narain Gehani, and William D. Roome. *The Concurrent C Programming Language*. Silicon Press, 1989.

[8] Gary A. Ford, and Richard S. Weiner. *Modula-2: A Software Development Approach*. John Wiley, 1985.

[9] Sape Mullender. *Distributed Systems*. ACM Press, 1990.

[10] Sape Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba-A distributed operating system for the 1990s. *IEEE Computer*, 23(5), May 1990, pp. 44-53.

[11] Willard Korfhage, and Ralph Yozzo. Picking processes for migration. In *Proceedings of the Fourth ISMM International Conference on Parallel and Distributed Computing and Systems*, Washington D.C., Oct. 1991.

[12] A. Nico Habermann, and Dewayne E. Perry. *Ada for Experienced Programmers*. Addison Wesley, 1983.

[13] Narain Gehani. *Ada: An Advanced Introduction*. Prentice Hall, 1983.

[14] Mamoru Maekawa, Arthur E. Oldehoeft, and Rodney R. Oldehoeft. *Operating Systems: Advanced Concepts*. Benjamin/Cummings, 1987.

[15] David F. Bacon, and Andy Lowry. A portable run-time system for the Hermes distributed programming language. In *Proceedings of Summer 1990 USENIX Conference*, Anaheim, CA, June 1990.