Proc. 2nd Int. Conf. on Protocol Specification, Testing, and Verification, INWG IFIP,
Ed. C. Sunshine, pp. 383-391, North-Holland Publ., Idyllwild, CA, May 1982.

ALGEBRAIC VALIDATION METHODS
a comparison of three techniques

Gerard J. Holzmann
Delft University of Technology
Dept. EE, Mekelweg 4, 2600 GA Delft
The Netherlands

1. INTRODUCTION

Protocol verification methods are commonly subdivided into two main groups. A distinction is then made between state-transition models and program-proving theories, with in between a small group of hybrid techniques [e.g. 3,7]. Our aim in this paper is to show that there is yet a third group of verification techniques that is slowly evolving into a main new discipline in its own right. This third group consists of the methods based on special validation algebras for communicating systems. Here we will briefly review three examples of techniques that belong to this emerging class.
Though we restrict the discussion to just these three techniques, there are also other validation methods [12,13,16], and indeed some specification methods [1,2,15] that share important concepts with them.

In section 2 we will consider Milner's calculus for communicating systems, CCS [10,11]. In section 3 we will discuss the Flow Expression Language introduced by Shaw [14], and finally in section 4 we will review the approach suggested in [4,5,6]. Section 5 summarizes our findings.

2. CCS — A CALCULUS FOR COMMUNICATING SYSTEMS [8-11]

Processes in CCS are modeled by "communicating agents" whose behaviors are described in algebraic expressions. CCS combines the usual operators from regular expression theory with a small set of operators that can be used to model concurrency and interaction. The proof rules from CCS are, however, quite different from the usual identities that hold for regular expressions. One of the major divergencies from these identities is the rejection of the distributive law for time orderings over summations in CCS. We will illustrate this with an example.

Consider the following two valid behavior expressions in CCS:

(a1.(a2 + a3)) and ((a1.a2) + (a1.a3))

In these expressions each symbol "ai" represents an action that can be performed by the agent modeled. The dot indicates a time ordering, the plus indicates a choice (referred to as a "summation of two behaviors"), and the parentheses determine the priority of evaluation. In CCS these two expressions are not considered to be equivalent, and hence the distributive law is rejected.

The reason for rejecting the distributive law is that it does not preserve the "observational equivalence" of behaviors. Informally, two behaviors are observationally equivalent if

they cannot be distinguished from one another by an outside observer who considers each behavior as a black box and only watches its external actions. The very informality of this definition, though, makes it suspect. The major part of CCS is, in fact, meant to establish more formalized notions of equivalence.
Even with the informal notion of equivalence, the observer would have no trouble distinguishing between the two behaviors in the above example. In the first case the system will be able to perform either an a2 or an a3 action after the initial a1 action. In the second case it can only perform one of these actions, and (unlike the first system) will reject all attempts to perform the other.
How the outside observer may be able to test the ability of the system to perform certain actions will become clear in the next paragraph.

Every "action" in CCS models an attempt to communicate with the outside world (possibly an observer, or some other agent). No action can be performed without communication. In other words: CCS only models the communication aspects of a system of processes and it abstracts from the rest. We will see below that in this aspect this method differs from the algebra suggested by Shaw in [14] (section 3) and resembles the algebra in [4-6] (section 4).

Communication between two agents can take place whenever they are able to perform "complementary actions." Each action has a unique name in CCS. The set of names known to each agent is called the agent's "sort." Corresponding to each name from that sort CCS defines a matching "co-name." The behaviors of two agents, then, are synchronized by the simultaneous action on a "name" in one agent, and on the corresponding "co-name" in the other. In this combined move a value (or a vector of values) may be passed from the agent that is acting on the co-name to the agent that is acting on the name itself.

Consider the following two expressions:

a(x).NIL and A(E).NIL

NIL is the representation of the empty set of possible continuations of the behavior, E is a value-returning expression (in the conventional sense), and x is a variable. In the above expressions a lower-case symbol (a) repre- sents an action name, and the upper-case equivalent (A) the corresponding co-name. (Milner uses symbols with an overscore for co-names.) The "sort" of both agents consists of just one name: a. As noted above, an agent cannot just perform actions at will, it can only perform an action if another agent will simultaneously perform a matching action on a co-name. In this case both agents can proceed by performing the only action in their behavior and then block on NIL. In the combined move on "a" the value of the expression given in E is passed from the second agent to the first, and delivered in variable x.

This mechanism for interactions is called the concept of the "synchronized communications." Clearly, it strongly resembles the rendez-vous concept in, for instance, Hoare's CSP language, and it may therefore be possible to develop CCS into a full proof language for CSP.
It is, however, important to note that in CCS a single name (or co-name) may occur in more than just one agent. An agent can synchronize with any single one from a group of other agents which can perform matching actions, without regard to the other agent's identity.

The main operations introduced in CCS to model concurrency and interactions are:

| (composition), which formalizes the concurrent execution of two behaviors (with possible interactions); general form: "(B1 | B2)," where B1 and B2 are behaviors.

\ (restriction), which is used to hide actions from an observer (or in the mathematical counterpart: to suppress terms in calculations); general form: "(B \ a)," where B is a behavior and "a" is an action-name.

/ (relabeling), which is used to match names in the sort of one agent to those of another, in order to make a communi- cation possible; general form: "(B[a1/a2]," where B is a general form: "(B[a1/a2])," where B is a behavior and a1 behavior and a1 and a2 are action-names. The effect is that all occurrences of a2 in B are renamed to a1.

The composition operation simply combines two behaviors by linking action names to co-names. If necessary a match can be enforced by first relabeling action-names.

One of the main theorems of CCS is the "expansion theorem," which defines how behavior expressions containing the composition operator can be elaborated into observationally equivalent expressions. In short, the expansion theorem formalizes a shuffling operation on individual behaviors. The expansion gives us the union (summation) of possible continuations of the composite agent. For each individual agent that can make a single move, there is one term in this union. (It is assumed that these moves will match external actions to be specified in later compositions.) There is, however, also one term in the expansion for each pair of agents that can perform matching actions in a single combined move. The combined move is represented by the greek symbol *t*. These *t* symbols are subsequently referred to as "hidden moves" or "internal communications."

Clearly, a full expansion of a composition according to the above rules would lead to an explosion of terms in the resulting expression. This is where the restriction operation comes in handy. With it we can suppress all moves on actions that we are not primarily interested in, e.g. the potential interactions with the outside world if we know that there won't be any such interaction.

A short example will clarify the above. Consider the composition:

((at + bt') | (Au + cu') | (Bv + Cv'))\ a \ b

In this composite no value-transfers are specified. Again lower-case letters are action-names, upper-case letters are co-names. Single actions on action-names "a" and "b" are hidden by the restriction operator. This eliminates the need for a full listing of the individual a-moves and b-moves in the expansion below.
The composite can be expanded into:

c(((u') | (at + bt') | (Bv + Cv'))\ a\ b)
+ C(((v') | (at + bt') | (Au + cu'))\ a\ b)
+ *t*(((t) | (u) | (Bv + Cv'))\ a\ b)
+ *t*(((t') | (v) | (Au + cu'))\ a\ b)
+ *t*(((u') | (v') | (at + bt'))\ a\ b)

The first two terms specify unrestricted actions on "c". The third term corresponds to a communication on action name "a", the fourth and fifth terms on action names "b" and "c", respectively.

We will restrict the discussion here to this small subset of CCS. The laws of CCS have been proven complete for expressions modeling terminating behaviors. (This means that any two behavior expressions that are observationally equivalent in any context can always be proven to be equal by these laws.)

CCS has, for instance, been used to analyze semaphore systems (with one process modeling a semaphore process) [11], and a distributed version of the dining philosophers' problem [9].

An early example of the application of (a subset of) CCS to the analysis of distributed data-base protocols can be found in [8]. In contrast to Robin Milner, however, George Milne uses broadcast communications and disallows value-transfers. In [8] Milne presents a complete (hand) derivation of a proof of freedom of deadlock for the data-base model. (This proof, though, covers 10 full pages.)

The notion of the "synchronized communications" of CCS, which may seem to be a handicap for its application in protocol studies, may turn out to be quite adequate for modeling communications on both very low and very high levels of message interaction. On a low level, a sender and a receiver can indeed only exchange a value if both are ready to engage in such an activity. On a high level the synchronized communication could be used to model (and subsequently abstract from) lower-level handshaking procedures on message exchanges.

3. LFE - A LANGUAGE OF FLOW EXPRESSIONS [14]

Unlike CCS, but similar to the algebra for protocol expressions from [4-6] (section 4), Shaw's language of Flow Expressions (henceforth referred to as LFE) can be defined as an extension of the language of regular expressions. The usual set of operators from regular expression theory (concatenation or dot product for time orderings, summation for choices, the Kleene star for repetition) is extended with a shuffle operator for interleaving two behaviors and special mechanisms for modeling interaction.

In CCS interaction was modeled by means of naming conventions for action symbols. We have noted above that no event can be modeled in CCS without some implied form of interaction. Quite the opposite point of view is taken in Shaw's LFE, where actions normally do not imply any synchronization at all. The purpose of the flow language is to model system entities in both sequential and concurrent environments. In the following, however, we will concentrate on those aspects of LFE that allow for the modeling of concurrent systems.

Shaw introduced two special symbols to model synchronizations: a wait symbol (ω) and a signal symbol (σ). The semantics are as follows. Performing an action with name "ω_i" means "wait for signal i to be generated." Performing an action with name "σ_i" means "generate signal i." The link between an ω and a σ is made via index i.

Where in CCS synchronization was modeled after the rendez-vous concept, synchronization in LFE seems to be modeled after Dijkstra's concept of the binary semaphore.

The synchronization operations do not allow for direct modeling of value passing in distributed systems.

Shaw defines separate language constraints for wait and signal actions. The main restriction imposed on these actions is that for each unique index "i" each wait action ω_i must be preceded by one or more σ_i actions. (Extra σ_i's are allowed.)

Consider the following composite (using the shuffle symbol from CCS):

$$((a.\omega_i.b) \mid (d.\sigma_i))$$

It can be expanded into:

$$(a.d.b + d.a.b)$$

The other terms of the full shuffle (e.g. (a.b.d)) are canceled by the synchronization constraint: $(\sigma_i.\omega_i)$.

It is quite easy to formalize verification problems in this flow expression language. For instance, reachability problems translate into membership problems: given an expression f is it within the language defined by L ? Similarly, a deadlock problem translates into the problem of determining whether the language L contains expressions that are not a prefix of any other expression within L. A deadlock in the initial state trivially becomes an emptiness problem: does L define the empty set of action sequences ?
In LFE there is no emphasis on equivalence problems, as in CCS, though it could possibly be extended in this way.

Flow expressions have, to our knowledge, not yet been applied to protocol analysis problems. A difficulty in such applications may be that LFE prescribes an asynchronous but memoryless type of synchronization. (In the first aspect it differs from CCS, in the second it differs from the algebra defined in [4-6].)
The communication is of a boolean type: a process is allowed to continue if at least one release (σ) signal was generated. Exactly how many release signals were generated is unknown. Shaw refers to a variant of LFE, the event expressions from [13], for a language that does have this capability. With this slight variation one can define "message transfer expressions" which can be used to model sequences of message interactions.

The description of concurrent systems in LFE is mainly directed towards manual analyses. One indication of this fact is that LFE defines no tools that can be used to prevent the explosive growth of the number of terms generated by the shuffling operator in loosely coupled systems.

4. PVA - A PROTOCOL VALIDATION ALGEBRA [4-6]

The main difference between the two validation theories discussed above has been the concept of synchronization. CCS is based on the rendez-vous concept, and LFE on the binary semaphore concept. We will see that the protocol validation algebra from [4-6] (from here on referred to as PVA) is based on the concept of buffered message exchanges. Still, disregarding minor notational differences, the three theories are quite close in their approach to the validation problem. Here we will concentrate on the differences that remain.

All communication in PVA is modeled in buffered message exchanges. Similar to CCS, though, the only thing that can be modeled is communication; local computations are treated as "non-events." In PVA, however, also value-transfers are treated as "non-events."
Abstraction from value-transfers is not an accidental "omission" in the PVA model. PVA consistently models generalities instead of specifics. This has the advantage that the results of a single analysis may hold for a whole class of protocols. Obviously, it also has the disadvantage that some protocol-specific properties cannot easily be analyzed.

In PVA each process has a "sort" of message names that can be accepted, but unlike CCS this sort only includes messages that can be received by a process, not those sent to others. Synchronization in PVA, then, is on mailboxes, not on message ("action") names.

Unlike both CCS and LFE, PVA introduces a special operator for sending. The observation that sending and receiving are really each others inverses has led to the choice for a fractional notation. Receive actions are written in the numerator; send actions are written in the denominator of a fraction. The following example will illustrate the advantages of this choice for the calculations with protocol expressions:

The act of sending a message "a" is represented in the sending process by the term: "(1/a)" and in the receiving process by the term "a". The cross-product (cf. "shuffle") of these sending and receiving actions (assuming that these are the only actions in the behavior of the sender and the receiver) leads to the following expansion in PVA:

$$
\begin{aligned}
((1/a) \times (a)) &= ((1/a).(a) + (a).(1/a)) \\
&= ((1/a).(a) + 0) \\
&= (1 + 0) \\
&= 1
\end{aligned}
$$

Apart from the two new operators (division and multiplication) there are two special symbols: 1 and 0. The symbol 0 is equivalent to the NIL symbol from CCS: it represents the empty set of possible continuations (cf. deadlock). The symbol 1 repre- sents the null string (cf. non-event, comparable to the symbol *t* in CCS).

On the first line of the expansion above we have simply elabo- rated the full shuffle of the two terms in the cross-product. The second equal sign is covered by a "soundness rule" within the algebra, the third is covered by a "reduction rule," and the last is an identity that holds for regular expressions in general.

The soundness rule from PVA is used to enforce proper mailbox orderings on shuffles. It rules out all terms in the shuffle that correspond to interactions where messages would be received in another order than the mailboxes hold them (i.e. fifo).

The reduction rule from PVA is similar in effect to the restriction operation in CCS. It can be used to abstract from series of message exchanges that we are not primarily interested in. It merely states that any "sound" message exchange may be replaced by the symbol 1. There is no comparable rule for restriction in LFE.

The concept of the message exchanges that forms the basis of the synchronization mechanisms in PVA has also prompted the inclusion of a timeout facility for receive actions. A timeout is a special message named "*t*" (not to be confused with the CCS symbol *t*) that can be appended to an empty mailbox by a system demon.
If a timeout option is specified the system demon can at any time append the timeout message, and thus force a restart of the waiting process. (The demon only requires that the mailbox be empty.)

It is fairly easy, for instance, to model communication over an unreliable channel, with timeout and retransmission in PVA.

User P is trying to communicate with user Q via channel C. P will send a packet "p", and wait for acknowledgment "a" from Q. If the acknowledgment does not arrive before P times out P will retransmit the packet. Channel C will delete, mutilate or transfer packets

between P and Q in some unpredictable way. The behaviors of P, Q and C can be represented as:

P: ((*t* + x + (a.1)) . (1/p))
C: ((p.(1/(1 + x' + p'))) + (a'.(1/(1 + x + a))))
Q: (x' + (p'/a'))

In the above system, transmission will start by P timing out on its empty mailbox (i.e. receiving *t* from the demon) and sending off the first packet "p" to channel C. C consumes the packet and either deletes it (symbol 1 in the sum), garbles it (i.e. transfers it as an unreadable packet "x'") or transfers it correctly to Q in "p'". Process Q will ignore any "x'" packets, but will acknowledge the "p'" packets with an "a'". If the "a'" packet makes it back through C, it will arrive as an "a" at P. Upon the receipt of the acknowledgement P will generate a new packet to be sent via "p". The act of generating new packets, however, is not detailed in PVA. In the above expression it is represented by a non-event 1, but it could equally well have been deleted since it is not relevant to the analysis.

As an aside, to prove that effective data transfer in the above modeled system is possible, it would be helpful if we could tag weights on terms in a summation. Clearly, the average number of times that a packet will have to be retransmitted depends on the probability that the channel will transfer it correctly. If this probability is non-zero we can at least establish an upper-bound to the number of retransmissions in P.
Probabilities, however, are not accounted for in either one of the three techniques discussed in this paper.

5. CONCLUSION

We have discussed three algebraic techniques that can be used for protocol validations. The discussion has revealed some important issues in the design of models for algebraic protocol validation. These issues can be summarized as follows:

(1) Value transfer
Are value-transfers an essential part of the validation problem (the standpoint taken in CCS), or should it be possible to abstract from them (LFE and PVA) ?

(2) Synchronization
On what abstraction level should we choose our paradigm for process synchronizations (see below) ?

(3) Proof theory
What should be the structure of the proof theory. Should the distributive law be included (LFE, PVA) or avoided (as in CCS) ?

(4) Automation
What specific tools should be included to allow for an efficient automation of the validation process ?

The three techniques we have reviewed resemble each other in the theoretical framework that they erect for studying interaction problems. Still, each technique offers a different set of answers to the four questions listed above. We have, for instance, noted that the basic concepts for modeling interaction are quite different: CCS's model of synchronization is derived from the rendez-vous concept, Shaw's LFE is based on a special type of

semaphore synchronization and PVA bases itself on synchronization via mailbox interactions.
Some of the basic rules of CCS, on the one side, and LFE and PVA, on the other, are found to be conflicting. In fact, it is readily possible to prove something in LFE or PVA that would be incorrect in CCS. The main cause for this difference is the rejection of the distributive law in CCS (see section 2). It is as yet an open question whether this difference will prove to be a crucial or a minor one.

CCS is the only one of the three models discussed here that includes a facility for modeling value-transfers, but we may add that this facility in CCS is not yet matched by a validation method for tracing the manipulation of the CCS variables.

In CCS and in PVA we found special mechanisms to prevent an explosive growth of the number of terms generated in shuffling operations. This would indicate that both CCS and PVA are good candidates for automated protocol validation processes. At present PVA seems to be the only technique in this class that has actually been automated. A set of programs that implements the validation process in PVA has been in use under a Unix timesharing system (Unix is a trademark of Bell Laboratories) for more than a year now. It runs on both a PDP 11/70 and a VAX 780 machine, and has even been ported to an 8-bit Z-80 microcomputer system running under the CP/M operating system (CP/M is a trademark of Digital Research).

In automated analyses we can test a protocol on its observance of a series of fairly standard correctness criteria (no deadlock, no residual messages, no potential timeout problems, completeness). Admittedly, more specific problems (flow control, message sequencing) are difficult to automate, but most probably also they are well within the range of manual algebraic validation methods.

REFERENCES

[1] G.V. Bochmann, Communication protocols and error recovery procedures. ACM SIGOPS Operating Systems Review, Vol. 9, No. 3, pp. 45-50, March 1975.
[2] G.V. Bochmann, Finite state description of communication protocols. Publ. No. 236, Dept. d'Informatique, University of Montreal, 55 pp., July 1976, (see pp. 35-37).
[3] G.V. Bochmann & C.A. Sunshine, Formal methods in communication protocol design, IEEE Transactions on Communications, Special Issue on Computer Network Architectures and Protocols. Vol. COM-28, No. 4, pp. 624-631, April 1980.
[4] G.J. Holzmann, The analysis of message passing systems, Bell Laboratories Computing Science Technical Report, CSTR No. 93, March 1981.
[5] G.J. Holzmann, An algebra for protocol validation, Proceedings INWG/NPL "Workshop on Protocol Testing - Towards Proof ?," pp. 377-391, May 1981.
[6] G.J. Holzmann, A theory for protocol validation, Accepted for publication in IEEE Transaction on Computers.
[7] P.M. Merlin, Specification and validation of protocols, IEEE Transactions on Communic., Vol. COM-27, pp. 1661-1680, (1979).
[8] G.J. Milne, Modelling distributed database protocols by synchronisation processes, University of Edingburgh, Computer Science Dept., Internal Report CSR-34-78, November 1978, 32 pages.
[9] G.J. Milne, The representation of communication and concurrency, California Institute of Technology, Pasedena, Ca 91125, Computer Science Dept. Report, Sept. 1980,

103 pages.
[10] R. Milner, An algebraic theory of synchronization, Lecture Notes in Computer Science, Vol. 67, 1979.
[11] R. Milner, A calculus for communicating systems. Lecture Notes in Computer Science, Vol. 92, 1980.
[12] W.F. Ogden, W.E. Riddle & W.C. Rounds, Complexity of expressions allowing concurrency. Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, pp. 185-194, Jan. 1978.
[13] W.E. Riddle, An approach to software system modelling, behavior specification and analysis, Dept. of Comput. and Commun. Sciences, University of Michigan, Ann Arbor, MI, RSSM/25, July 1976.
[14] A.C. Shaw, Software descriptions with flow expressions. IEEE Transactions on SE, Vol. SE-4, No. 3, pp. 242-254, May 1978.
[15] S. Schindler, Algebraic and model specification techniques. Proc. 13th Hawaii Intern. Conference on System Sciences, pp. 20-34, Jan. 1980.
[16] A.Y. Teng & M.T. Liu, A formal approach to the design and implementation of network communication protocols. Proc. COMPSAC 1978, IEEE, pp. 722-727.