Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras*

Egon Börger
Dipartimento di Informatica
Università di Pisa
56125 Pisa, Italy
boerger@di.unipi.it

Uwe Glässer
Heinz Nixdorf Institut
Universität-GH Paderborn
33098 Paderborn, Germany
glaesser@uni-paderborn.de

July 19, 1995

Abstract

This is a tutorial introduction into the evolving algebra approach to design and verification of complex computing systems. It is written to be used by the working computer scientist. We explain the salient features of the methodology by showing how one can develop from scratch an easily understandable and transparent evolving algebra model for PVM, the widespread virtual architecture for heterogeneous distributed computing.

Introduction

In 1988 Yuri Gurevich has discovered the notion of *evolving algebra* in an attempt to sharpen Turing's thesis by complexity theoretic considerations (see [22]). Through numerous case studies (see [4] for an annotated list which is complete up to 1994) it has become clear since then that using the notion of evolving algebras one can develop a powerful and elegant specification methodology which has a huge yet unexplored potential for industrial applications. In this report we are going to explain the basic concepts of this approach to the design and analysis of complex systems and illustrate its salient features through a challenging example from real life.

We provide an abstract formal specification of central components of PVM, the widespread virtual architecture for heterogeneous distributed computing [20, 32]. We start from scratch without presupposing any knowledge of PVM;

*In BRICS TR (BRICS-NS-95-4), University of Aarhus, July 1995. Preliminary versions of this paper have been presented by U. Glässer at the PVM Users' Group Meeting, Oak Ridge, TN, May, 1994 and by E. Börger at the Euro–PVM Users' Group Meeting, Rome, Oct., 1994.

the fact that we will nevertheless end up with a clear and easily understandable formal definition of the main features of PVM's *virtual machine* at the level of the user interface[1] is not the worst argument for the specification methodology.

It is for the benefit of the practitioner that we are going to introduce here the notion of evolving algebra by a non–toy example from real–life; for a precise mathematical foundation we refer the interested reader to [23]. Our exposition is centered around what we consider to be the four outstanding merits of the proposed approach to the specification and verification of complex systems:

- the *freedom of abstraction* which allows you to structure a system into a hierarchy of appropriate subsystems,
- the powerful and simple mechanism for *information hiding* and definition of *precise interfaces*,
- the principle of *locality for dynamics* (state transitions), and
- the satisfactory link to application domains by the construction of *appropriate ground models*.

1 Freedom of Abstraction

It is well known that general abstraction principles are needed to cope with the complexity of large systems. For data structures the algebraic specification approach (see [33]) shows a way to deal with abstract data types; for actions the action semantics approach (see [30]) proposes a scheme for constructing complex operations out of basic components. Evolving algebras offer the possibility to choose both, the data and the basic actions, at any level of abstraction and independently of each other. The way how this is done is very simple and corresponds to common practice in systems engineering: when specifying a software or hardware system one has to define its basic *objects* and the elementary *operations* which the system uses for its actions (dynamical behaviour). In other words one has to define the basic domains and functions of a system. This leads in a natural way to the mathematical notion of *structures* as formalization of system *states*, as we are going to explain now.

1.1 Universes

Each system S deals with certain basic *objects* which might be classified into different categories. This is reflected in an evolving algebra model of S by corresponding *sets* (also called universes or domains), one for each category of objects. These sets can be completely abstract—this is the case if no restriction

[1] More precisely, we specify the C–interface of the virtual machine of PVM 3, the current version of PVM [19].

is imposed on the corresponding category of objects. In case that the objects are assumed to have certain properties or to be in certain relations with other objects, we formalize these properties and relations by corresponding conditions (integrity constraints) which the objects in those sets are required to satisfy. The evolving algebra approach accepts any precise formalization of such conditions, in whatever language or framework they are given.

In the remaining part of this subsection we illustrate this data abstraction principle by a discussion of basic PVM domains.

Under PVM a heterogeneous collection of physically interconnected Unix–based machines of various kinds of architectures (including serial, parallel, and vector computers) appears logically as a loosely coupled distributed–memory computer. The constituting member computers are called *host* machines and are formalized as elements of a corresponding set $HOST$. The architecture of each host machine is indicated by a function

$$arch : HOST \rightarrow ARCH$$

where $ARCH$ is the set of possible architectures to be used with PVM 3 as listed in [19]. An important feature which distinguishes the universe $HOST$ from $ARCH$ is that the latter is static—i.e. it does not change—whereas the former is dynamic. Indeed, host machines can be dynamically added to or deleted from the virtual machine—except for a designated host

$$master : HOST$$

on which PVM is started and which maintains the control over the dynamically changing machine configuration. Since the main intention of the concept of evolving algebras is to reflect the dynamical system behaviour in a direct and simple way, there are two basic operations which formalize the growing and shrinking of universes:

$$\textbf{extend } A \textbf{ by } x_1, \ldots, x_n \textbf{ with } — \textbf{ endextend}$$

where '—' is used to define certain properties or functions for (some of) the new objects x_i of the universe A. For example, if a new host machine of architecture type at has to be added—this can be obtained in PVM using the routine `pvm_addhosts()`—one can write

$$\textbf{extend } HOST \textbf{ by } x \textbf{ with } arch(x) := at \textbf{ endextend} .$$

The corresponding deletion operation has the following form

$$\textbf{discard } t \textbf{ from } A$$

where t is a first–order term[2] and A a universe. One might be tempted to assume that once t is deleted from A, each function where t appears as value or in the arguments is automatically undefined there; but since this is more a type–checking or implementation concern (about error detection or garbage collection) than a semantical concern, we assume here only that '**discard** t **from** A' has the effect of setting the characteristic function of A at t to false.

The basic computational units of PVM are constituted by concurrently running application programs (Unix processes), which can enroll into PVM as *tasks*. We formalize these tasks as elements of a dynamic domain

$$TASK \subset PROCESS$$

which is required to consist of the processes enrolled into PVM[3]. When a process enrolls into PVM (see below the PVM instruction `pvm_mytid()`) it is registered by a new task identifier, i.e. an appropriate set TID—which is left abstract here—is extended by a new element which is assigned to the given process, say $Process$, by an injective function tid as follows:

$$\textbf{extend } TID \textbf{ by } Tid \textbf{ with } tid(Process) := Tid \textbf{ endextend}$$

Note that also the inverse function:

$$task : TID \to TASK$$

of tid is used. By requiring $task$ to be the inverse function of tid we avoid to have to explicitly set $task(TID) := Process$ when $tid(Process)$ is set to Tid. This formalizes the global addressing of tasks in PVM.

On the basis of this homogeneous global address space TID in which all tasks are uniformly addressed through tid, when tasks communicate to each other they do not need to know whether their communication partner resides on the same host or not. Indeed, the asynchronous message–passing model of PVM does not distinguish between local and global intertask communication. However, on each of the hosts there is a local daemon process, called *pvmd*, which acts as a local supervisor in operations that require task management or intertask communication. We formalize this by introducing another dynamic set $DAEMON$ together with a nullary function $pvmd : DAEMON$ identifying the concurrently operating PVM daemons. The correspondence between daemons and their hosts is expressed by a dynamic bijective function:

$$host : DAEMON \to HOST.$$

[2] By *first–order term* we mean any expression built up from constants or variables by using function symbols.

[3] Note that there are also other processes which are not running under PVM, for instance, the PVM system itself is such a process.

Clearly, each time a host is added, a new daemon has to be created and the function *host* to be updated correspondingly. Thus the above $HOST$–extension is refined as follows:

extend $HOST$ **by** x **with**
  $arch(x) := at$
  extend $DAEMON$ **by** d **with**
    $host(d) := x$
  endextend
endextend

For the initialization we have to require that there is a distinguished daemon

$$demiurge : DAEMON$$

who resides on the master host (i.e. $host(demiurge) = master$) and will in particular be responsible for creation and deletion of hosts.

1.2 Dynamic Functions

Once it has become clear what are the basic objects of a system S, one has to think about what are the elementary operations which are performed on those objects in S. Typically, a basic operation consists of setting a certain value, given the values of certain parameters. The most general framework of such operations is the following *function update*:

$$f(t_1, \ldots, t_n) := t$$

where f is an arbitrary n–ary function and $t_1, \ldots, t_n$ represent the parameters (arguments) at which the value of the function is set to t. In writing evolving algebras it is allowed to use arbitrary function updates, for functions f and terms t_i, t of arbitrary complexity or level of abstraction. Note that such functions f are called *dynamic* in contrast to *static* functions which do not change.

Function updates provide the basic notion of *destructive assignment at any level of abstraction*. Here are two very simple and well known but characteristic examples for function updates. The updating of a program counter in an architecture is expressed by the following function update

$$pc := next(pc)$$

where pc represents a 0–ary function and $next$ the monadic function which determines for a given value of pc what is the next value to be stored in pc. The updating of a buffer by adding a new datum can be expressed by the following function update:

$$cont(buffer) := append(datum, cont(buffer))$$

where *buffer* is a 0-ary function, *cont* a monadic function providing the buffer content and *append* an abstract function which given a datum and a buffer content yields the result of appending this datum to the current content of the buffer.

Besides domain extension and deletion of elements, function updates are the mechanism by which the dynamics of arbitrary systems can be described in an explicit way. In accordance with usual practice the execution of updates in evolving algebras can be conditioned by so–called guards, giving rise to so–called *transition rules* (also called *guarded multi–update instruction* in [23]) of the following form:

if *Cond* **then** *Updates*

Cond is an arbitrary boolean valued expression (first–order logic formula) and *Updates* a set of updates. The rule can be executed if *Cond* is true; its effect is to simultaneously execute each update in the set *Updates*. (The simultaneous execution of more than one update helps to avoid an explicit description of intermediate storage, see for example the two updates $a := b, b := a$.)

A simple example from PVM is provided by the routine `pvm_mytid()`. If this routine is called by a process *Process*, the responsible *pvmd* checks whether *Process* is among the tasks which are registered already—the set of those tasks is stored by a dynamic function *tids* at argument *pvmd*. If *Process* is already registered, its task identifier is returned to *Process*; if it is not registered yet, then it gets enrolled into PVM as described in the preceding subsection and its task identifier is added to $tids(pvmd)$. Thus we have the following rule:

if $tid(Process) \neq undef$
then *return* $tid(Process)$ *to Process*
else extend *TID* **by** *Tid* **with**
$tid(Process) := Tid$
$tids(pvmd) := insert(Tid, tids(pvmd))$
return Tid to Process
endextend

Here "*return t to P*" is an abstract update which we are not going to specify furthermore at this level of abstraction.

1.3 States as Algebras

To speak about a system means to speak about its objects in terms of functions and relations defined on them. Domains, functions, and relations constitute what in mathematics is called a *structure*. Structures without relations are traditionally called *algebras*. Since relations (and in particular sets) can be represented by their characteristic functions, for simplicity, we deal in the following only with algebras.

An *evolving algebra* can (in a first approximation) be defined as a finite set of transition rules

if *Cond* **then** *Updates*

where *Updates* consists of finitely many function updates (and domain extensions or deletions of elements; these two update forms can be reduced to function updates, see [23]). The effect of a transition rule R when applied to an algebra $\mathcal{A}$ is to produce another algebra $\mathcal{A}'$ which differs from $\mathcal{A}$ by the new values for those functions at those arguments where the values are updated by the rule R.

We will construct the PVM rules in such a way that the guards imply consistency of simultaneously executable updates. Note that no rule changes the type of the functions; only the incarnation (the concrete interpretation) of a function changes by changing some of its values. We speak therefore of algebras also as *static* algebras, to distinguish them from evolving algebras. The latter are transition systems which transform the former.

Thus, the abstraction principle which is built into the notion of evolving algebra consists in proposing

- (static) algebras as the mathematical notion of "state" and
- guarded destructive assignments for abstract functions as basic dynamic operations

This is the most general notion of state and of dynamic changes of states modern mathematics offer. As a consequence evolving algebras are the most general notion of a (discrete) dynamic system. A priori no restriction is imposed on the abstraction level where one might want to place an evolving algebra description of a system. This freedom explains the success of the simple and transparent evolving algebra models for the semantics and the implementation of numerous complex programming languages like Prolog [15, 16], C [24], VHDL [11], Occam [9, 8], for protocols [12, 27], architectures [6, 5, 13], real–time algorithms [25, 26], etc.

The importance of the freedom of abstraction which is offered through the notion of evolving algebra is also confirmed by a common experience in the design of algorithms. Namely, the need to model phenomena of the real world, which are given a priori, leads the designer of programs to use 'abstract structures', as has been well expressed a long time ago by N. Wirth:

> *"... Data in the first instance represent abstractions of real phenomena and are preferably formulated as abstract structures not necessarily realized in common programming languages."* (see [34], page 10)

Furthermore, the reciprocal dependency of algorithms and data structures makes it important for the designer not to be hindered by inappropriate restrictions of the framework; to say it again with Wirth's words:

> *"It is clear that decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often depend strongly on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably interwined."* (see [34], page 9)

The framework of evolving algebras offers the freedom the designer needs to 'tailor' his models to the given level of abstraction.

2 Information Hiding and Interfaces

The basic idea of *information hiding*, as introduced by D. Parnas in [31], addresses the modular structuring of systems and can be summerized as follows:

> *"A module achieves program simplification by providing an abstraction. That is, its function can be understood through its interface definition without any need to understand the internal details."* [29].

In a practical specification methodology information hiding has to go hand in hand with a good discipline to handle interfaces. The evolving algebra approach offers both in a most general way through the concept of *external functions*.

Each function f which appears in an update $f(t_1, \ldots, t_n) := t$ of some transition rule R is called internal for R; for a given evolving algebra $\mathcal{A}$ a function g of the vocabulary of $\mathcal{A}$ which is not internal for any of the rules of $\mathcal{A}$ is called external for $\mathcal{A}$.

For a function f which is internal for $\mathcal{A}$ all the information on (the dynamical behaviour of) f is available through the rules of $\mathcal{A}$; the programmer who knows the rules of $\mathcal{A}$ can use that information about f. In contrast, for an external function g for $\mathcal{A}$ the rules of $\mathcal{A}$ give no information on the behaviour of g. An external function g cannot be modified ('written') by $\mathcal{A}$, but it can be used ('read') in the rules of $\mathcal{A}$ to determine arguments at which an internal function is changed dynamically or to determine the new values in such updates. External functions can in particular be used to represent the environment in which an evolving algebra is intended to work. It is the task of the system designer to provide exactly that information on g which he wants the programmer to know and to use. In the evolving algebra methodology this interface information can range from nothing at all—this is the case of an external function for which only the number and the types of its arguments and values are known—to a full specification by some axioms or by a set of equations or by another evolving algebra (module), etc. Note that due to the abstraction principle explained in the previous section the evolving algebra approach imposes no restriction at all on the choice of external functions and the way they are described. The use of evolving algebras does not trivialize the difficult task of "designer control of the

distribution of information" ([31]:pg. 344), but at least it does not hinder this task by extraneous formal overhead and offers a flexible and open framework to guarantee information hiding and the definition of precise abstract interfaces.

In the following subsections we explain some outstanding examples for the use of external functions in evolving algebra descriptions of complex systems.

2.1 The PVM Event Mechanism

PVM realizes a distributed computation model which is characterized by the *reactive behaviour* of the concurrently operating PVM daemon processes, one on each host computer of the virtual machine. The daemons effectively carry out the PVM instructions of their local tasks and may interact with each other through asynchronous message–passing communication. There are in principle two different kinds of operations requiring the activity of a daemon (pvmd): a request of a local task to carry out some PVM instruction and the reception of a message from another pvmd. The daemon cannot influence from where, when, and which request or message will reach him, rather he has to wait for the next such event to come whenever he is idle. We can model this intuition faithfully by introducing an external function *event* which for some given pvmd might yield a PVM instruction or message as value. If $event(pvmd)$ is defined and has the value *instr/mssg*, then the pvmd is going to execute/read *instr/mssg*. This is formalized in our PVM model by a rule of form

if $event(pvmd) = instr/mssg$ **then** $execute\_instr/read\_mssg$

for each individual PVM instruction *instr* or PVM message *mssg*, where *execute_instr/read_mssg* represents the corresponding updates. An integrity constraint on the function *event* is that a defined value of $event(pvmd)$ remains stable until the pvmd has evaluated the function. However, we assume 'destructive reading' such that $event(pvmd)$ is reset to *undef* (resp. indicates the next event) as soon as the pvmd has read the current value.

As example we define the four rules which define the task administration and administration of message buffers in PVM.

2.1.1 PVM Task Administration

We complete the formalization of the PVM routine `pvm_mytid()` by the following rule defining the reaction of the pvmd when the routine is called by a given process *Process*:

`pvm_mytid()`

if $event(pvmd) = mytid()$ *from Process*
thenif $tid(Process) \neq undef$
then *return* $tid(Process)$ *to Process*
else extend *TID* **by** *Tid* **with**
$tid(Process) := Tid$
$tids(pvmd) := insert(Tid, tids(pvmd))$
return Tid to Process
endextend

Similary, we have the following rule for the PVM routine `pvm_exit()` by which a *Task* can leave PVM through a request to the responsible pvmd:

`pvm_exit()`

if $event(pvmd) = exit()$ *from Task*
then
discard $tid(Task)$ **from** *TID*
$tids(pvmd) := delete(tid(Task), tids(pvmd))$

2.1.2 PVM Message Buffers

For the specification of PVM's message–passing interface—which offers point-to-point communication from one task to another as well as multicast to a set of tasks—we extend the basic model by abstract domains related to messages and buffers: $MESSAGE$, TAG, $DATA$, $BUFID$, $ENCODING$. To send a message, a task first packs the message into a send buffer and then calls one of the send functions. To selectively receive messages, a task invokes one of various receive functions specifying the *receive context*. Though PVM does not explicitly specify any limit to the size or number of messages, our specification could easily incorporate conditions which reflect constraints coming from physical limitations of the underlying hardware and software components.

The *basic* message-passing routines of PVM apply a simple communication model[4] that is based on two fundamental assumptions: for each task there is only one send buffer and one receive buffer; any message transfer between tasks is handled by the responsible pvmds. The following description of the message-passing model assumes two basic integrity constraints that are guaranteed by

[4] or more sophisticated communication mechanisms additional routines and options, not discussed here, allow to tailor the basic model to individual application requirements; for example, this includes direct task-to-task communication, specific group functions, or multiple send and receive buffers.

the corresponding routines embedded in the virtual machine: message-passing is *reliable* and *order-preserving*.

Message buffers are addressed through unique identifiers from $BUFID$, $0 \notin BUFID$. The content of a buffer may be any sequence of $DATA$ objects and is accessed by $cont : BUFID \rightarrow DATA^*$. An $encoding : BUFID \rightarrow ENCODING = \{PvmDataDefault, PvmDataRaw, PvmDataInPlace\}$, associated to send buffers, specifies the method used for packing messages: *PvmDataDefault* refers to External Data Representation (XDR); *PvmDataRaw* to the original data format; *PvmDataInPlace* means that the data items have to be copied directly out of the user's memory (for details cf. [19]). Two injective functions *sendbuf* and *recvbuf* from $TASK$ into $BUFID$ yield the current send and receive buffers of tasks.

The routine *pvm_initsend*() creates, for the task which wants to start a sending operation, an empty send buffer with the specified encoding scheme and returns a corresponding buffer identifier. A class of routines *pvm_pk**$(Pointer, Nitem, Stride)$—there is one pack routine for each individual data type $* \in \{byte, cplx, dcplx, double, float, int, long, short\}$[5]—packs the number *Nitem* of data into the send buffer; *Pointer* refers to the location of the first data item and *Stride* to the relative distance to the next one. The formal definitions of these routines are given by the rules:

`pvm_initsend()`

if $event(x) = initsend(Encoding)$ *from Task*
then
  extend $BUFID$ **by** b **with**
    $sendbuf(Task) := b$
    $cont(b) := \langle\rangle$
    $encoding(b) := Encoding$
    return $\langle b \rangle$ *to Task*
  endextend

`pvm_pk*()`

if $event(x) = pk^*(Pointer, Nitem, Stride)$ *from Task*
then
  $cont(b)) := append(data^*(Pointer, Nitem, Stride), cont(b))$

where $b \equiv sendbuf(Task)$

We will complete the formalization of the PVM message–passing system in Section 3.

[5]For packing strings a simpler routine is used which we do not describe here.

2.2 Further Examples of External Functions

The external function *event* in our PVM model is dynamic and thus directly reflects the way in which tasks interact with their local pvmd when they want PVM routines to be invoked. No pvmd has an own fixed PVM program; the instructions or messages which reach a pvmd come for him as determined by his environment to which he reacts. Using the external dynamic function *event* we abstract from the specific way how the daemon's walk through his sequence of instructions/messages is determined by the activities of his tasks.

The power of abstraction which is offered through the introduction of external functions does not depend on their being dynamic. Consider as example the formal definition given in [15] for the programming language Prolog as seen by the programmer. The four simple rules which define the full behaviour of Prolog for user–defined predicates make crucial use of two external functions *procdef* and *unify*.

The function *procdef* is supposed to provide for a given literal l and the given program db exactly the clauses in db which are relevant for l in the order in which they have to be applied. The whole non–trivial backtracking behaviour of Prolog (including optimizations like determinacy detection) can be described on the basis of such a function *procdef* without being more specific about the latter. *procdef* plays for that model the role of a "well–defined whole." If we consider Prolog without operations like *assert, retract* which modify the program, then *procdef* is a static external function. If we want to model also Prolog's program modification features then *procdef* becomes an internal dynamic function (see [7, 14]). Note that during the refinement process by which the Prolog model of [15] is linked in a provably correct way to the WAM implementation model in [16] the function *procdef* receives an explicit definition.

The function *unify* is supposed in [15] to provide for each pair of literals either a unifying substitution or the information that there is no such unification. This function describes the abstract behaviour of unification without being bound to any concrete unification algorithm. It also hides from the programmer the details about the representation of termes which appear in the refined WAM models of [16]. As a result the abstract PROLOG model of [15] and its refinement to the WAM model of [16] could easily be extended to constraint logic programming languages with or without types where unifiability appears as a particular case of constraints (see the formal evolving algebra definitions of PROLOG III [18], Protos–L [1], and CLP(R) [17]).

Other examples of external functions which contributed in a crucial way to the simplicity of the models under consideration are the following:

- the static find–catcher function defined in [15] leads to a concise formalization of the error–handling predicates *catch* and *throw* of Prolog.
- In the evolving algebra model for the IEEE VHDL Standard we have obtained a simple and uniform rule set for signal assignments by intro-

ducing for the inertial delay an external static function *reject* for which we give a natural and easily understandable recursive definition. Similarly, a transparent description is obtained for the propagation of signal values by introducing external functions for the so–called driving and effective values; the former is determined by a recursion on the signal sources, the latter by a recursion on port association elements from ports to signals. In both cases the recursive definitions replace rather complex algorithmic characterizations in the VHDL'93 language reference manual [28].

- In the abstract evolving algebra models of Occam (see [9]) which are the starting point for the correctness proof of a compilation scheme into Transputer instructions in [8] we have taken great advantage of the usual flowchart layout of programs; we define it by external functions which in the later refinement steps are replaced by recursive definitions of the compiling function. Considerable simplifications for both the specifications and the proofs have also been obtained there by leaving the evaluation and compilation of expressions and the implementation of values abstract, realized by appropriately restricted external functions.

- In the more theoretical example constituted by Lamport's mutual exclusion protocol, known as bakery algorithm, a tremendous simplification of the correctness proofs in the literature has been achieved in [12] by introducing two external functions, namely *Ticket* and *Go*, on which three natural conditions and an induction principle are imposed which imply the correctness of the protocol.

In all these cases the external functions allowed us to define a precise interface with respect to which the model under discussion works in a simple and transparent way. Clearly, if for such an abstract model we want to prove general properties about the behaviour of the system where external functions play a role, we have to state and assume their properties which we use. In order to guarantee an unchanged interface behaviour these properties have to be proved to be satisfied when the external functions are defined explicitly or implemented in later refinement steps or modified by changing requirements.

The most general *concept of modularity* which is present through the notion of external functions is deliberately kept open in the definition of evolving algebras. The resulting flexibility in using and dealing with different module structures is an advantage for real–life specification endeavours. Nothing prevents us from restricting this notion to specific and even syntactic concepts of compositionality where the need arises; an example where it turned out to be useful to stick to a simple automaton–theoretic concept of composition of evolving algebras through sequencing, juxtaposition, and feedback can be found in [5].

3 Locality for Dynamics

It is typical for large systems that their overall dynamical behaviour is determined by the dynamic behaviour of their components, i.e. by *local* changes. The evolving algebra methodology allows us to reflect this characteristic interplay between global and local dynamic system behaviour in a direct and faithful way by viewing (global) system states as static algebras and by providing the possibility to express local updates $f(t_1, \ldots, t_n) := t$ in a uniform way at any level of abstraction. This becomes particularly evident with attempts to model distributed systems. We are going to illustrate this through a formalization of the PVM message–passing interface.

Interactions between Pvmds Certain PVM instructions result in distributed operations involving two or more pvmds which perform some interaction through message–passing. We model inter–pvmd communication in the transition rules below by abstract updates of the form

'*forward* ⟨*RequestMsg*⟩ *to* y' respectively '*return* ⟨*ReplyMsg*⟩ *to* x'

where x and y refer to the interacting pvmds. They are supposed to trigger corresponding events for the receiving pvmds. For that reason we will have a number of communication related transition rules, distinguished by the suffix *'_req_msg'* for request messages and *'_rep_msg'* for reply messages, describing the reaction of a pvmd x when receiving a message from another pvmd y.

Distributed operations in which a pvmd requests another pvmd for service may produce considerable delays. In order to avoid blocking while waiting for the response to come, the requesting pvmd stores the "wait context" using a structure accessed by a unique *wait-id (wid)* from a domain *WID*. This wait-id is passed along with the request and returned with the reply. The wait context typically includes information about the requesting task ($req\_info : WID \rightarrow TASK$), the reply data ($rep\_info : WID \rightarrow REPLY^*$), and a request count ($waitcount : WID \rightarrow INT$, indicating the number of replies a pvmd is still waiting for).

The preceding concepts suffice to formalize the status information related to subroutines of PVM, as we are going to do in the next subsection.

3.1 PVM Status Information

PVM offers various kinds of routines providing status information about the virtual machine. To obtain the status of a host, a call of *pvm_mstat*() yields: *PvmOk* if the host is running, *PvmHostFail* if it is unreachable, or *PvmNoHost* if this host is not in the virtual machine. Since a host failure is to be considered as an external event, outside the scope of PVM, the resulting effect is modeled using an external dynamic function $hstatus$. When triggered by a requesting

pvmd, *hstatus* does not immediately yield a result, as it usually takes some time to perform the distributed operation required to determine the status of a remote host. After a delay—corresponding to the duration of the simulated operation—*hstatus* replies by generating an *ext_hstatus* report event, returning the requested status information to the calling pvmd.

Note that an interaction between a pvmd and the external function *hstatus* appears to the pvmd almost the same way as an interaction with a remote pvmd. That is, it also requires to create a wait context structure, the wid of which is passed over in the function call and returned with the *ext_hstatus* report: $hstatus : HOST \times WID \rightarrow \{running, unreachable\} \times WID$.

To find out whether a particular host is in the virtual machine, a pvmd simply performs a look up on the host table. This test is expressed through the predicate $is\_in\_htable : HOST \rightarrow BOOL$:

`pvm_mstat()`

if $event(pvmd) = mstat(Host)$ *from* $Task$
thenif $is\_in\_htable(Host)$
  thenif $Host = host(x)$
  then
    return $\langle PvmOk \rangle$ *to* $Task$
  else
    extend WID **by** wid **with**
      $rep\_info(wid) := Task$
      trigger $hstatus(Host, wid)$
    endextend
else *return* $\langle PvmNoHost \rangle$ *to* $Task$

`ext_hstatus_report`

if $event(pvmd) =$
  $ext\_hstatus\_report\langle Status, Wid \rangle$
thenif $Status = running$
  then
    return $\langle PvmOk \rangle$ *to* $Task$
  else
    return $\langle PvmHostFail \rangle$ *to* $Task$

where $Task \equiv rep\_info(Wid)$

We can now also explain the send and receive instructions of PVM.

3.2 PVM Send Routines

A message consists of receiver and sender tid, an integer tag, and the actual message data; these components are accessible through functions defined on $MESSAGE$ with values in TID ($recvtid$, $sendtid$), TAG ($msgtag$), $DATA^*$ ($msgdata$). Thus each pvmd x holds a sequence $msgseq(x)$ of messages waiting—in the order of their arrival at x—to be received by one of its local tasks. An additional function $msgbuf : MESSAGE \rightarrow BUFID$ is used to hold the addresses of local buffers which are used by the pvmd to store the message data until these messages are delivered to the receiving tasks.

The routine $pvm\_send(Tid, Tag)$ puts the information, stored in the send buffer of the sending task, into a newly created *message* that is sent to Tid with label Tag. If the daemon of the receiver task is the same as that of the the sender task, sending the message in principle means to enqueue it into the daemon's message queue; otherwise, the message will be forwarded as intertask–message to the remote pvmd identified through Tid (causing a corresponding event for that pvmd).

Each time the pvmd x receives a message, it has to check whether the receiver task has specified a *receive context* that matches the labels of the message. The receive context of a task is expressed by a function $expecting : TASK \rightarrow \langle (TID + \{-1\}) \times (TAG + \{-1\}) \rangle$ as a combination of options for the message Tag and sender Tid, where '-1' indicates matching every possible combination. If for some given task t of pvmd x $expecting(t)$ has a defined value, this means that t has been *suspended* on an attempt to receive a message which was not yet available; that is, a message of the specified type was not yet contained in $msgseq(x)$.

Note in the definition of `pvm_send()` that the message data are simply stored in a local buffer in case that sender and receiver work under the same pvmd; otherwise, they are stored as part of the message using the function $msgdata$. This meaning is formally defined by the following rules:

`pvm_send()`

if $event(pvmd) = send(\,Tid\,,\,Tag\,)\ from\ Task$
then
  extend $MESSAGE$ **by** m **with**
    $recvtid(m) := Tid$
    $msgtag(m) := Tag$
    $sendtid(m) := tid(\,Task\,)$
    if $pvmd(\,Tid\,) = pvmd$
    then
      $Enqueue(m, msgseq(pvmd), Type, Data, Receiver)$
    else
      $msgdata(m) := cont(sendbuf(\,Task\,))$
      $forward\ \ intertask\_msg\langle m\rangle\ \ to\ pvmd(\,Tid\,)$
  endextend

  where $\ Type \equiv \langle tid(\,Task\,),\,Tag\rangle,$
    $Data \equiv cont(sendbuf(\,Task\,)),$
    $Receiver \equiv task(\,Tid\,)$

$Enqueue(mssg, mssgs, type, data, receiver)$

$\equiv$ **extend** $BUFID$ **by** b **with**
    $cont(b) := data$
    $msgbuf(mssg) := b$
    if $matching(type, expecting(receiver))$
    then
      $recvbuf(receiver) := b$
      $expecting(receiver) := undef$
  endextend
  $mssgs := append(mssg, mssgs)$

`intertask_msg()`

if $event(pvmd) = intertask\_msg\langle m\rangle\ \ from\ pvmd'$
then $Enqueue(m, msgseq(pvmd), Type, Data, Receiver)$

where $\ Type \equiv \langle sendtid(m), msgtag(m)\rangle,$
  $Data \equiv msgdata(m),$
  $Receiver \equiv task(recvtid(m))$

The corresponding multicast routine *pvm_mcast*() can be handled in almost the same way.

3.3 PVM Receive Routines

PVM supports blocking and non–blocking receive routines. Blocking *recv*() requested from *Task* sets the receive buffer of *Task* to the bufid of an expected message and returns bufid to *Task*, if an appropriate message is actually waiting in the message queue; otherwise, the given receive context is used to update the *expecting* function of *Task*. Non–blocking *nrecv*() does the same except for returning 0 to *Task* instead of suspending the task, in case there is no message of the expected type in $msgseq(x)$. Formally this is expressed by two rules, where the function

$$\lceil \quad : \quad MESSAGE^* \times \langle TID \times (TID + \{-1\}) \times (TAG + \{-1\}) \rangle \rightarrow MESSAGE^*$$

selects, out of $msgseq(x)$, the messages corresponding to *expecting*:

`pvm_recv()`

if $event(pvmd) = recv(Tid, Tag)$ *from* *Task*
thenif $msgseq(pvmd) \lceil \langle tid(Task), Tid, Tag \rangle = [\,]$
then $expecting(Task) := \langle Tid, Tag \rangle$
else $RecvMsg(Tid, Tag, Task)$*Through pvmd*

`pvm_nrecv()`

if $event(pvmd) = nrec(Tid, Tag)$ *from* *Task*
thenif $msgseq(x) \lceil \langle tid(Task), Tid, Tag \rangle = [\,]$
then *return* $\langle 0 \rangle$ *to* *Task*
else $RecvMsg(Tid, Tag, Task)$ *Through pvmd*

$RecvMsg(Tid, Tag, Task)$ *Through pvmd*

$\equiv$ **let** $msgseq(pvmd) \lceil \langle tid(Task), Tid, Tag \rangle = [m|Mssgs]$
return $\langle bufid \rangle$ *to* *Task*
$recvbuf(Task) := bufid$
$msgseq(pvmd) :=$ *delete* m *from* $msgseq(x)$

where $bufid \equiv msgbuf(m)$

We conclude this section with two significant examples for the formalization of process control constructs in PVM, namely the creation and the deletion of tasks.

3.4 PVM Task Creation

The routine pvm_spawn() enables dynamic subtask creation. On initiating a spawn operation the spawning *task* becomes *parent* of the (*Ntask* many) new subtasks to be created, each of which is assumed to run a copy of a given executable *File* (possibly with a list of *Arg*uments). The spawning task may affect the selection of hosts to spawn on through a mode parameter: in *transparent* mode tasks are automatically executed on the most appropriate hosts w.r.t. certain load measures; in *architecture-dependent* mode the calling task specifies the architecture; in *low-level* mode it specifies a particular host.

Parameters *Flag* and *Where* are used to specify a combination of options as a sum of: *0 PvmTaskDefault*—PVM chooses where to spawn the processes; *1 PvmTaskHost*—the *Where* argument specifies a particular host to spawn on; *2 PvmTaskArch*—*Where* specifies a type of architecture to spawn on using ARCH; *4 PvmTaskDebug*—starts these processes up under debugger; *8 PvmTaskTrace*—the PVM calls in these processes will generate trace data[6].

In assigning tasks to hosts upon spawning, PVM's in principle non–deterministic choice depends in particular upon the environment (e.g. the operating system) and the internal load balancing scheme (which is transparent to the user). We abstract from details of this complex selection procedure by using a dynamic *external* function *hostselect* which we assume, as integrity constraint, to be consistent with the options set by the user (through *Flag, Where, Ntask*). For the result of *hostselect*, which is a list of the form

$$\langle pvmd_0, n_0\rangle, \quad \langle pvmd_1, n_1\rangle, \ldots, \langle pvmd_m, n_m\rangle$$

specifying a collection of pvmds and the number of tasks to be spawned by each of these pvmds, we require that the sum of the n_i, $i = 0, \ldots, m$, is less than or equal to *Ntask* and that $\forall i, j \in \{0, \ldots, m\} : \; i \neq j \Rightarrow pvmd_i \neq pvmd_j$.

The outcome of an attempt to spawn a task which has been assigned to a host depends on the availability of resources, and on whether a suitable copy of the relevant *File* is present on that host. We model the behaviour of the system using again a dynamic external function *try_to_spawn* which provides the necessary *SPAWNREPORT*. Since computing this function may be rather time consuming and thus is not an atomic action, spawning has two phases, modeled by separate rules: a spawn operation is triggered by calling *try_to_spawn* with the appropriate argument values computed using *hostselect*. On completion of this operation *try_to_spawn* returns, to the calling pvmd, the resulting list of pids, one for each successfully created process required to spawn a new task, by generating an *ext_spawn_report* event.

[6] This feature is not yet implemented, cf. [19].

`pvm_spawn()`

if $event(pvmd) = spawn(File, Args, Flag, Where, Ntask)$ *from* $Task$
thenif $hostselect(Flag, Where, Ntask, pvmd) = \langle local, remote \rangle$
& $local = \langle pvmd, n_0 \rangle$
& $remote = \langle pvmd_1, n_1 \rangle, \ldots, \langle pvmd_m, n_m \rangle$
then
let $tid = tid(Task)$
extend WID **by** wid **with**
spawning_task(wid) := tid
tasks_to_be_spawned(wid) := $\langle\rangle$
$waitcount(wid) := m + signum(n_0)$
if $n_0 > 0$
then *trigger* $try_to_spawn\langle$Params$, n_0, wid\rangle$
var i **ranges over** $\{1, \ldots, m\}$
if $m > 0$
then *forward* $spawn_req_msg\langle$Params$, n_i, wid, tid\rangle$ *to* $pvmd_i$
endextend

where Params$\equiv File, Args, Flag, Where$

`ext_spawn_report`$\langle\rangle$

if $event(pvmd) = ext\_spawn\_report\langle Wid, \langle Pid_1, ..., Pid_n\rangle\rangle$
then
  let tid = spawning_task(Wid)
  extend TID **by** $tid_1, \ldots, tid_n$ **with**
    let $Tids = tid_1, \ldots, tid_n$
    extend $TASK$ **by** $t_1, \ldots, t_n$ **with**
      $\vdots$
      $pvmd(tid_j) := pvmd$
      $Enroll(t_j, tid_j, tid, Pid_j)$
      $\vdots$
    endextend
    if $pvmd(tid) = pvmd$
    then $SpawnRepInfo(Tids, Wid)$
    else *return* $spawn\_rep\_msg\langle Tids, Wid\rangle$ *to* $pvmd(tid)$
  endextend

  where $Enroll(Task, Tid, Parent, Pid)$
      $\equiv$ $tid(Task) := Tid$
        $pid(Task) := Pid$
        $parent(Task) := Parent$

When receiving an *ext_spawn_report*, the pvmd enrolls the reported processes as tasks. Depending on whether the task that has initiated the spawn request is local or not, the resulting list of new tids either is appended to the local wait context (in *SpawnRepInfo*) or it is returned to the corresponding remote pvmd. We omit the straightforward formalization of *SpawnRepInfo*($Wid, Tids$) : the returned wait identifier Wid provides access to the local wait context where the request counter $waitcount(Wid)$ and the list *rep_info(Wid)* of tids of successfully spawned tasks are updated; upon final completion, the result of the distributed spawn operation is returned to the calling task $req\_info(Wid)$.

The rules for interaction between the spawning task pvmd and the pvmds on selected remote hosts are:

`spawn_req_msg`$\langle\rangle$

if $event(pvmd) = spawn\_req\_msg\langle Params, Wid, Tid\rangle$ *from* $pvmd'$
then
  extend *WID* **by** *wid* **with**
    $req\_info(wid) := Tid$
    $rep\_info(wid) := Wid$
    trigger $try\_to\_spawn\langle Params, wid\rangle$
  endextend

where $Params \equiv File, Args, Flag, Where, N$

`spawn_rep_msg`$\langle\rangle$

if $event(pvmd) = spawn\_rep\_msg\langle Tids, Wid\rangle$ *from* $pvmd'$
then $SpawnRepInfo(Tids, Wid)$

Remark Note that the function *hostselect* used above is a particular interesting example which shows the integration potential of the evolving algebra methodology. This function is an external dynamic function for the PVM model constituted by our rules. It contains however a static subfunction which is defined by conditions on host selection which are given in the PVM manual.

3.5 PVM Task Deletion

The routine $pvm\_kill(Tid)$ causes the pvmd to kill the task identified by Tid. In the formal description a symbolic system command, *kill_process*, is used to express the resulting required interaction between PVM and the operating system. If the task to be killed resides on a remote processor, the local pvmd forwards a kill message to the corresponding remote pvmd. Upon receiving a kill_msg() a pvmd acts as if it had received a pvm_kill by one of its local tasks.

`pvm_kill()`

if $event(pvmd) = kill(Tid)$ *from* *Task*
thenif $pvmd(Tid) = pvmd$
then
  delete *Tid* *from* *TID*
  $kill\_process(pid(task(Tid)))$
else
  forward $kill\_msg\langle Tid\rangle$ *to* $pvmd(Tid)$

`kill_msg`$\langle\rangle$

if $event(pvmd) = kill\_msg\langle Tid\rangle$ *from* $pvmd'$
then
  delete Tid from TID
  $kill\_process(pid(task(Tid)))$

4 Appropriate Ground Models

As has been observed in [3, 21] evolving algebras fit in a special way when it comes to link a non–formal system description S (requirements specification) to a formal model S_0 which has to be recognized as faithful formalization of S. This is because one can tailor the level of abstraction of the formal model S_0 to be built in such a way that its domains and functions directly correspond to the basic objects and operations of S; this correspondence must be understandable by inspection because by definition there is no possibility to prove any correctness statement about S. Further refinements of the model S_0, once this *ground model* has been recognized as "correct", are obtained through possibly formal transformations leading to an implementation; since at this level only formal models are involved, in the ideal case the correctness of the transformations can be proved (by mathematical or theorem proving methods).

For an interesting although small example of such a system see the evolving algebra specification of the steam–boiler control which has been refined leading from the ground model to a running C^{++} program (see [2]). A more involved example is constituted by the problem of faithfully defining the standard for a programming language, i.e. in an easy to understand but precise and complete way. For the ISO Prolog Standard published in 1995 and the IEEE VHDL'93 Standard evolving algebra ground models appear in [15, 11].

5 Distributed Evolving Algebras

In this section we resume the preceding discussion by the definition of distributed evolving algebras, taken from [23] to which we refer for details.

A *distributed evolving algebra* $\mathcal{A}$ comes as a finite set of sequential evolving algebra programs, called *modules*, together with a finite set of concurrently operating *agents* executing these modules, and a collection of *initial states* of $\mathcal{A}$. The vocabulary of $\mathcal{A}$ is obtained by combining the vocabularies of the sequential evolving algebra programs of all modules. A function *Mod* specifies the mapping from agents to modules where the latter are identified through *module names*. In contrast to the function *Mod* and the set of agents, which may both change dynamically, module names are static nullary functions. In addtion to *Mod* there

is a special nullary function *Self* which each agent interprets in a different way: an agent a interprets *Self* as a.

A distributed computation of $\mathcal{A}$ starting in some initial state S_0 of $\mathcal{A}$ and running through a sequence of states $S_1\ S_2\ \ldots$ is defined by the collective actions of the various agents. (Note that each agent a has its own partial view of a given state S_i as reflected by the vocabulary of $Mod(a)$.) In a single computation step leading from state S_i to state S_{i+1} one or more agents may participate such that each of these agents contributes to the local state changes by making his own *move*. Since the moves of the subcomputations of the individual agents are linearly ordered in time, a *run* of $\mathcal{A}$ in principle is characterized by a partially ordered set of moves together with the associated agents and the resulting states. A rigorous semantical definition of the notion of runs of distributed evolving algebras is given in [23].

The Distributed *PVM Algebra* In our PVM model the agents are represented by the concurrently operating PVM daemons as identified through the function *pvmd* (taking the role of *Self*). The set of pvmds increases or decreases depending on the dynamically changing configuration of the virtual machine. As there are only two different kinds of pvmds, the *demiurge* on the master host (see Section 1.1) and the ordinary pvmds on the other hosts, we have only two different types of modules. For the the PVM features that have been discussed here, the differences in the functional behaviour of these two types of pvmds are irrelevant.

6 Conclusions

Continuing the work started in [10] we have defined a formal model for PVM from scratch. The goal was to illustrate some outstanding merits of the evolving algebra methodology to design and analysis of complex computer systems. This text has been written for the practitioner; we advice the theoretically inclined to study [23] where Gurevich provides a rigorous mathematical foundation of the semantics of evolving algebras.

References

[1] Ch. Beierle and E. Börger. A WAM extension for type-constraint logic programming: Specification and correctness proof. Research report IWBS 200, IBM Germany Science Center, Heidelberg, December 1991.

[2] Ch. Beierle, E. Börger, I. Đurđanović U. Glässer, and E. Riccobene. An evolving algebra solution to the steam-boiler control specification problem. Seminar on *Methods for Specification and Semantics* (Dagstuhl, June 1995), Report, 1995.

[3] E. Börger. Logic Programming: The Evolving Algebra Approach. In B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam, 391–395.

[4] E. Börger. Annotated bibliography on evolving algebras. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[5] E. Börger and G. Del Castillo. A formal method for provably correct composition of a real–life processor out of basic components (The APE100 reverse engineering project). In *Proc. of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*.

[6] Egon Börger, Giuseppe Del Castillo, Paola Glavan, and Dean Rosenzweig. Towards a mathematical specification of the APE100 architecture: The APESE model. In B. Pehrson and I. Simon, editors, *Proc. of the IFIP 13th World Computer Congress 1994, Volume I: Technology and Foundations*, pages 396–401. Elsevier Science Publishers B. V., 1994.

[7] E. Börger and B. Demoen. A framework to specify database update views for Prolog. In M. J. Maluszynski, editor, *PLILP'91. Third International Symposium on Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1991.

[8] E. Börger and I. Đurđanović. Correctness of compiling Occam to Transputer code. 1995 (submitted).

[9] E. Börger, I. Đurđanović, and D. Rosenzweig. Occam: Specification and compiler correctness. part i: The primary model. In E.-R. Olderog, editor, *Proc. of PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.

[10] E. Börger and U.Glässer. A formal specification of the PVM architecture. In B. Pehrson and I. Simon (Eds.) *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam, 402–409.

[11] E. Börger, U. Glässer, and W. Mueller. Formal definition of an abstract VHDL'93 simulator by EA–machines. In C. Delgado Kloos and Peter T. Breuer, editors, *Semantics of VHDL*, volume 307 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1995.

[12] E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[13] E. Börger and S. Mazzanti. A correctness proof for pipelining in RISC architectures. Manuscript, 1995.

[14] E. Börger and D. Rosenzweig. An analysis of Prolog database views and their uniform implementation. Research report CSE-TR-89-91, EECS, University of Michigan, Ann Arbor MI, 1991.

[15] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 1995.

[16] E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In L. C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Series in Computer Science and Artificial Intelligence. Elsevier Science B.V./North–Holland, 1995.

[17] E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP($\mathcal{R}$) programs. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[18] E. Börger and P. Schmitt. A formal operational semantics for languages of type Prolog III. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 67–79. Springer, 1991.

[19] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, September 1994.

[20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994. to appear.

[21] U. Glässer. Systems Level Specification and Modelling of Reactive Systems: Concepts, Methods, and Tools. In Proc. of the *Fifth International Conference on Computer Aided Systems Technology (EUROCAST'95)*, Innsbruck, Austria, May, 1995.

[22] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[23] Yuri Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[24] Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[25] Y. Gurevich, J. Huggins, and R. Mani. The Generalized Railroad Crossing Problem: An Evolving Algebra Based Solution. EECS Department, University of Michigan–Ann Arbor, CSE Technical Report, CSE-TR-230-95, 1995.

[26] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[27] Jim Huggins. Kermit: Specification and verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[28] The Institute of Electrical and Electronics Engineering. *IEEE Standard VHDL Language Reference Manual—IEEE Std 1076–1993*, New York, NY, USA, 1994.

[29] . M. Marcotty and H.F. Ledgard. The World of Programming Languages. Springer-Verlag, 1986.

[30] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[31] David L. Parnas. Information distribution aspects of design methodology. In C. V. Freiman, editor, *Proc. of IFIP Congress 1971, Volume 1: Foundations and Systems*, pages 339–344. North-Holland, 1972.

[32] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20(4):531–545, 1994.

[33] Martin Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B*, pages 675–788, Elsevier, 1990.

[34] Niklaus Wirth. Algorithms & Data Structures. Prentice–Hall, 1975.