A Superimposition Control Construct for Distributed Systems

SHMUEL KATZ
The Technion

A control structure called a *superimposition* is proposed. The structure contains schematic abstractions of processes called *roletypes* in its declaration. Each roletype may be bound to processes from a basic distributed algorithm, and the operations of the roletype will then execute interleaved with those of the basic processes, over the same state space. This structure captures a kind of modularity natural for distributed programming, which previously has been treated using a macro-like implantation of code. The elements of a superimposition are identified, a syntax is suggested, correctness criteria are defined, and examples are presented.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**] Concurrent Programming; D.3.3 [**Programming Languages**] Language Constructs and Features—*control structures*; *modules*, *packages*

General Terms: Design, Languages

Additional Key Words and Phrases: Distributed programming, control construct, formal and actual processes, modularity, roletype, superimposition

1. INTRODUCTION

A control construct called a *superimposition* is proposed in order to capture a type of decomposition appropriate for distributed systems. In this procedure-like control structure, both formal processes and schematic abstractions of processes called *roletypes* are declared, each with formal parameters and a sequential communicating algorithm using those parameters. The declaration captures a distributed algorithm that is separately designed, but is intended to be executed in conjunction with other activities in the same state space. The construct is combined with an existing collection of communicating processes by instantiating the formal processes and associating each process of the collection with one roletype. The actual parameters connect the formal superimposition roletype to the state of the actual process. During execution the operations of the roletype declaration are interleaved with those of the

This work was supported in part by the Argentinian Research Fund at the Technion under grant 120-749.

Author's address: Technion, Israel Institute of Technology, Computer Science Department, Haifa 32000, Israel.

actual processes. Thus, the roletype can be seen as additional code to augment each actual process with which it is associated.

A schematic diagram intended to indicate the combination of a superimposition declaration with an existing set of *basic* processes is shown in Figure 1. At the left of the figure are shapes that represent both roletypes and formal processes. After combination of the superimposition and basic processes (represented as circles), the combination has each basic process augmented by one of the roletypes, and the formal processes of the superimposition added as additional processes of the combination.

Before turning to a fuller examination of the syntax and operational semantics of the construct, including scoping rules and correctness criteria, some motivation seems called for. A superimposition is intended to provide one of several concepts of modularization appropriate for the design of distributed systems. These concepts should aid in the decomposition of a system specification to more manageable subunits, supplementing the usual groupings such as blocks, conditionals, and procedures seen in sequential programming.

The first, and most obvious, decomposition of a system design is to isolate into a module a collection of processes operating in parallel with the remainder of the system. Some processes in the collection accept request messages from outside the collection, and eventually some processes in the collection send messages outside as appropriate responses. The necessary computations occur in parallel to the other processes of the system. This "vertical" division into groups of processes is the natural way to treat, for example, distributed implementations of data structures.

However, as noted already by Lamport [20], the type of decomposition described above is inappropriate for the many situations in which a modularization is required that cuts across process boundaries. A second type of decomposition defines what can be termed a *layer* of execution. In this case, a collection of already existent processes temporarily cooperate in some pattern of interaction in order to accomplish a subtask, and then continue in separate computations. Once a process begins executing the code of the layer, it will not execute operations that are not in the layer until it has finished its role within the layer. This type of modularization was first suggested in the *communication-closed layers* of [11] and extended somewhat in [15].

Both the *script* mechanism in [14], and the *team* of the Raddle language [12] are syntactic mechanisms for isolating interaction patterns among collections of processes. In these constructs, processes fill roles, which can be viewed as formal parameters of the script or team, and then return to their normal operation. A script thus amounts to an n-way subroutine mechanism. Note that while a script is executing, the participating processes are executing only operations of the script, and no other operations of those processes are interleaved.

Neither of the above types of decomposition are appropriate for a class of examples that has emerged over the last decade. Thus, *superimposition*, a third category of modularization, is needed. In this category, the module is an algorithm designed to be executed in processes also performing other tasks,

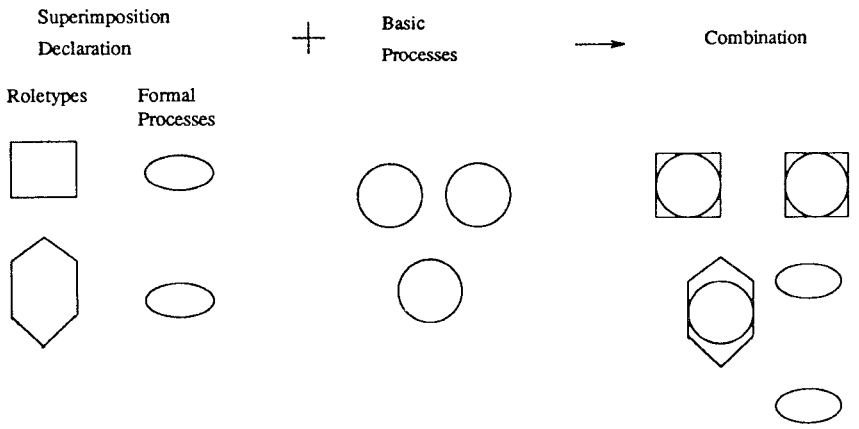

Fig. 1. A schematic combination.

with the operations of the algorithm interleaved with the operations for the other tasks. The initial specification is thus divided into two subtasks, one to be solved by a *basic* algorithm, and another by an algorithm to be interleaved with the basic algorithm, over the same state space.

An early use of the term "superimpose" is in the termination detection algorithm of Dijkstra and Scholten [8]. Numerous examples are explicitly intended to be used as superimpositions, starting with the termination detection algorithm of Francez [13] for nondiffusing computations, algorithms for deadlock detection ([7] and many others) and the global snapshot of [4]. All of these instances were developed and presented without considering the general issue of superimposition. They are characterized by the separate description of the algorithm to be superimposed and a rather informal description of the way in which the algorithm is to interact with the other tasks being performed in the processes. The sometimes implicit assumption is also made that the result of superimposing is a complete interleaving of the basic computation and the superimposed code.

More recently, the notion of superimposition (but termed superposition) has been investigated by Chandy and Misra in the context of their Unity approach to programming [5, 6]. A view of superimposition with somewhat different properties than the version seen here has been developed independently by Bouge and Francez [3]. Refinements used to distribute a centralized action system are also viewed as superimpositions in [2]. A detailed comparison among the approaches to superimposition is made in Section 5.

As may be seen from the examples in the literature, superimposition is particularly appropriate for adding functionality to a system in stages. For example, a distributed system might be required to be deadlock-free, to eventually terminate once it reaches an "inactive" global state, to guarantee eventual service to every request, and to guarantee that certain sections of code not execute concurrently, while at the same time executing some basic computation with a functional specification of its own. Each of these tasks

can be accomplished by superimposing an algorithm on a basic computation (which might in itself be the result of a previous superimposition on an even more basic algorithm). Superimposition is also appropriate for describing more mundane tasks such as monitoring, accounting, and debugging other algorithms.

In the remainder of this paper, the superimposition of one algorithm on others is examined in greater detail, and a language construct to isolate a superimposition in a separate program module is defined and illustrated. In Section 4, the generic proof of a superimposition with respect to its specification is considered, as are the proof obligations when an instance of a superimposition is combined with a basic computation.

In the continuation, a *superimposure*, or *combination*, will be considered as the result of merging a *basic* algorithm (or computation) and a new algorithm, called the *superimposition*. The merge allows an interleaving of the operations of the two algorithms and establishes a correspondence between the processes of the basic computation and the roletypes of the superimposition. Each process and roletype that are associated are, at least conceptually, operating on the same local state.

In order to keep the superimposition declaration independent of any specific basic algorithm, when the combination is made the superimposition must be adapted to the architecture of the basic algorithm. Moreover, any required adjustments in the basic algorithm should be described uniformly in the superimposition declaration. The degree to which such adjustments are allowed is discussed in Sections 4 and 5.

Although the ideas presented here can be applied for any model or type of program, for the sake of concreteness, *distributed reactive* systems [16] will be treated. These are distributed systems where each variable is local to some process, and values are passed between processes using explicit communication commands. A process is organized in a top-level loop that examines a variety of possibilities in guarded commands, where each possibility has a condition for execution (called a *guard*) determined by the values of local variables and including possible communications among processes. When a guard in a process is true, it may be selected ("passed"). The process then reacts by receiving a possible message and then executing associated local statements (including possible sending of messages if sending is nonblocking, and therefore local). Execution then returns to the top level. A guard and the associated local statements are called a *step*. Invariants and other temporal assertions are all expected to be true at the top level of the loop, before the selection of a guard for execution. Interleaving of basic and superimposed computations will mean that additional steps will (at least conceptually) be added by the superimposition to the steps of the basic computation. A finer interleaving, of every atomic statement, would have similar definitions but be more difficult to implement and verify. Note that if a shared-memory model were being treated, it might have been necessary to view several statements collected into a step as an atomic unit of execution in order to guarantee that a consistent state is maintained. In the distributed model, this is not necessary: any delay between the execution of local statements that together make

up a step and their (conceptual) interleavings with executions of statements from other processes do not affect the correctness of the algorithms. A process P involved in a step cannot do any other step until it has completed all of the associated local statements, and statements executed in any other process cannot affect the state of P until it has completed all local statements of the step—because all blocking communication and global tests are done at the beginning of the step. (A formal discussion of when execution sequences of distributed programs can be considered equivalent for the purposes of correctness can be found in [18].)

The normal form [1] of a CSP program is a reactive system. Also, in the Raddle model [12] the processes have such a reactive top-level organization and are composed of a collection of *rules*. In the context of superimposition, the basic rules are transformed and additional high-level rules are added. This terminology will be used in the continuation. When general control constructs are used in the basic algorithm, a mechanism is needed to link the superimposition actions with (abstract) locations in the basic program. This is briefly treated in Section 2.

2. A LANGUAGE CONSTRUCT FOR SUPERIMPOSITION

In an explicit superimposition language construct, a syntax is required both for the *declaration* of a superimposition and for a *combination* operator, in which a basic algorithm and a superimposition declaration are associated in order to produce a superimposure (i.e., an augmented algorithm). The superimposition declaration remains as a module of a program in the language. This is done for the same reasons that procedures are used in sequential programming: the code is isolated and thus easy to modify, renaming local variables is treated automatically by the scoping rules, parameter passing increases the generality of the module, and code need not be expanded.

Although the specification of a superimposition needs to relate to the state space of the basic computation, it would clearly be too restrictive to require identifiers in the basic computation with particular names fixed by the superimposition in advance. Instead, it is sufficient to use the parameter mechanism to specify that *some* variable in the basic algorithm corresponds to each formal variable in the superimposition and that it satisfies appropriate conditions. In order to further increase this generality, the formal process names appropriate for the superimposed code need not be (and usually are not) the same as those of the processes of the basic computation. Since each process of the basic algorithm is to be augmented, it is natural to define several *roletypes*, such as *initiator*, *follower*, *gatherer*. In the combination operator, each process of the basic computation will be associated with one roletype, and variables or constants of the process are bound to the parameters of the roletype. The specification of the superimposition may include restrictions on the processes (or the number of processes) that may be of each roletype. The formal processes of the superimposition declaration are instantiated with actual variables in place of the formal parameters, as additional processes in the combination.

Operationally, this means that each basic process is augmented with code "serving" the superimposition, and then during execution control is occasionally transferred from the code of the basic process to that of the roletype, the parameters are passed, one rule is tried (if it is enabled it will be executed), and control is returned to the actual process, with the parameters updated.

How often the control should be transferred and which rules should be tried are questions of *fairness*. It has been left open whether the rules in a superimposition are automatically guaranteed to execute due to a fairness assumption, or whether this guarantee must be coded into the transformations and rules. It seems desirable to maintain internal consistency in any programming language that adopts superimposition. Thus, whatever is true about the fair choice of rules in a process (or, in general, whatever is true for choices among alternative actions) should also hold for the interpretation of a superimposition under the macrosubstitution view. As a minimal requirement, within finite time the formal roletype should be activated. If no guard is then enabled, control is returned to the actual process without performing any action, while otherwise any one of the enabled rules of the roletype is chosen. This will guarantee what is known in [21] as *justice* for the roletype. It would also be possible to require that each rule of the superimposition that is continuously enabled is eventually executed, giving justice at the level of the rules. This is similar to the fairness condition of [6].

A superimposition declaration consists of type declarations, regular process declarations, and a collection of roletype declarations. There may also be a **global result** section stating a specification for the entire algorithm and a **global condition** giving global applicability conditions.

Each roletype contains a header and possible (sub)sections. The header consists of the keyword **roletype** followed by a name, a formal parameter list, and some restrictions. The sections of a roletype are a **var** section containing declarations of variables local to the roletype, a **transform** section describing the syntactic augmentation of the processes of the basic computation bound to that roletype, an **initialize** section with code to be executed when the combination operator is invoked, a **rules** section to express the part of the new algorithm to be (conceptually) interleaved with the code in basic processes of that roletype, and a **finalize** section to be executed when the basic algorithm is terminated. Each of the sections are optional, and will not appear if not needed for that roletype. In addition, each roletype may have a **result** section containing the result assertions about the combination for each process of that roletype and a **condition** section for the applicability conditions of the associated process from the basic computation that are not amenable to a syntactic transformation.

The outline of a typical superimposition declaration is seen in Figure 2. The (global and local) applicability conditions and the result assertions in this construct may be seen as comments not affecting the implementation. They are part of the specification of a superimposition, but will be given informally in the examples here. Their role in defining correctness will be seen later, in Section 4.

It should also be noted that although the emphasis here is on the code associated and interleaved with existing processes using the roletype mecha-

```
superimp <supername>
<type-declarations>
global result
      {result assertion for the superimposition}
global condition
      {global applicability condition}
   roletype <rolename> ( <parameter list> ) {, <restrictions>}
         var <local declarations>
         result
            {result assertions for this roletype}
         condition
            {applicability conditions for processes to use this roletype}
         transform
            {description of transformations to the basic computation}
         initialize
            {code to be executed before the basic computation begins}
         rules
            {rules for this roletype}
         finalize
            {code to be executed after the basic computation is completed}
   endroletype
   roletype ...
      .
      .
endsuperimp
```

Fig. 2. The form of a superimposition.

nism, some new processes could be added as part of the superimposition. With the simplifying assumptions given in the previous section, the declaration simply has the word **process** in place of **roletype** and does not have **transform** or **finalize** sections. In general, the syntax for adding new processes depends on the language into which the superimposition construct is to be incorporated. Section 3.1 has a straightforward example of adding a process.

The parameter list in the header of each roletype represents the formal interface to the processes of the basic algorithm to which the roletype will be bound. The transformations and rules are expressed in terms of these parameters. Thus, indirectly, through the binding done by the combination operator, these parameters connect the superimposition to the state space of the basic computation. Any variable in the superimposition declaration that is not a parameter must be assigned values within the superimposition code and does not return values to the basic algorithm.

The scope of all variables in the declaration, both parameters and any other variables in the superimposition, is local to the superimposition. That is, even if there are variables of the same name in the basic computation, they are unrelated to those in the superimposition declaration. It is as if a consistent renaming had occurred in either the superimposition declaration or in the basic algorithm.

The binding mechanism that seems most reasonable for a superimposition is call-by-reference (i.e., associating the address of the actual parameter with the formal parameter). If well-known restrictions on aliasing are obeyed [10]

(the same actual parameter cannot be associated with more than one formal parameter), the call-by-reference mechanism as used in the **rules** section is equivalent to a call-by-value-result. That is, each time the superimposed algorithm is to be activated, it may be considered as if the value of the actual parameter in the basic computation is transferred to the formal parameter in the superimposed algorithm, a rule is executed, and the value of the formal parameter is transferred back to the actual parameter in the appropriate process.

The restrictions after a role name involve the number of processes of that type in the resultant superimposure (e.g., *unrestricted*, *atmost 1*, or *atleast 1*) or structural requirements (e.g., *single-neighbor*).

If the basic programs do not have the simple loop structure assumed so far, it is possible to use the parameter mechanism to indicate which operations of the roletype are to be associated with which locations in the basic process. Just as for variables, a label parameter is used to represent "abstract" locations associated with the possible choice of indicated activities. These locations can be restricted in the **conditions** section. When a combination is created, labels from the basic process must be provided satisfying the restrictions. This approach is only reasonable if a generic identifier or comment can be used to indicate the type of label required at the time the combination is made (e.g., *choice-point*, *after-loop*, or *end-of-loop-body*).

The **transform** section contains those changes and additions to the basic algorithm that cannot be expressed as independent rules because they are associated with particular events in the basic algorithm. Such changes are difficult to describe uniformly, and sometimes [3] it is assumed that the basic algorithm has already been hand-tailored to suit the needs of the superimposition. However, that assumption makes the superimposition declaration incomplete and requires the designer of the basic algorithm to be aware of the requirements of the superimposition.

Here the description of transformations to the basic algorithm is in the form of rewrite rules. A pattern is followed by the keyword **to** and another pattern that includes new program statements. If the first pattern is only to be augmented with additional code, a "*" is used to indicate the pattern identified. These may be viewed as a modification to the BNF description of the syntax of a program. The intended meaning is that in parsing the basic algorithm, the new expression to the right of the keyword *to*, with the formal parameter names replaced by the actual parameter names, and * replaced by the identified pattern to the left of **to**, should be used instead of the identified pattern. The effect of these transformations is to syntactically modify the original basic algorithm. Thus the transformation

$$\langle \text{assignment} \rangle \ \textbf{to}^*; \ \text{count} := \text{count} + 1$$

means that each assignment statement will be augmented by incrementing the local variable *count*. This mechanism is potentially dangerous because arbitrary changes to the basic computation can be introduced in this section. Thus it is reasonable to limit the type of permissible transformations. Which transformations should be allowed is considered in greater detail in Section 5.

As an alternative to using transformations, when only additional statements are added at crucial locations of the basic program, the abstract labeling mentioned above can be used with the **rules** section.

A superimposure is formed by combining a group of processes that constitute the basic algorithm with a superimposition declaration. To summarize the effect of such a combination, the resultant superimposure has a process for each explicitly declared process of the superimposition declaration and an augmented process for each process of the basic algorithm. For the augmented processes, a correspondence is established between the processes of the basic algorithm and instances of the roletypes of the superimposition, and actual parameters are provided for each instance. The syntax for producing superimposures again depends on the facilities in the language for grouping processes. For simplicity, here, the correspondence is established by writing the line

use ⟨supername⟩

after the name of a group of processes that constitute the basic algorithm, and then after the heading of each basic process, the line

include ⟨supername⟩.⟨rolename⟩ (⟨actual parameter list⟩).

The actual parameter list gives the variables of that process that are to be associated with the formal variables in the declaration. There, clearly, is a syntactic check that all of the restrictions of the superimposition are satisfied. For a new process of the superimposition, the **include** line only serves to instantiate parameters and is not associated with a basic process. (It uses variables in the actual parameters from the scope at that point.) Note that several superimpositions on the same basic computation are possible, but they are not necessarily commutative. The resultant superimposure should be seen as a new collection of processes, and the original basic algorithm is still available if needed.

3. EXAMPLES OF SUPERIMPOSITIONS

3.1 Monitoring Execution

The text in Figure 3 concisely expresses a trivial superimposition to gather statistics on the number of assignments executed in a basic algorithm. Generalizations of the same idea should be useful for bookkeeping and debugging. In this version the compilation of information gathered is not superimposed on an existing process of the basic algorithm, but occurs in an additional "monitoring" process.

Note that it is not far from an informal description: it is assumed that only one assignment is executed per step. Each participant process increments a local variable each time an assignment is executed. The count is sent to the monitoring process every c assignment statements (by making the regular guards *false*, the superimposed code becomes the only thing left to do). Then the local variable is reset to zero. When the basic computation has finished

```
superimp monitoring
    global result when done, assignsum.monitorcoord = sum of assignments executed in basic algorithm
    global condition single assignment per step

roletype participant (c:integer )
    var count: integer;
    result count = sum of assignment statements in the process since last report
    transform <assignment> to * ; count:=count+1
              <guard> to count≠c; *
    initialize count := 0
    rules
       [count = c  →  send count to monitorcoord;
                      count := 0]
    finalize
        send count to monitorcoord;
        send done() to monitorcoord
endroletype {participant}

process monitorcoord(N: integer)
    var assignsum, t, donecount: integer; continue: boolean;
    initialize assignsum := 0; donecount := 0; continue := true;
    rules
       [continue; donecount < N, receive t from participant  →  assignsum  = assignsum + t]
       [continue; donecount < N; receive done() from participant  →  donecount = donecount+1]
       [continue; donecount = N  →  "report"; continue  = false]
endprocess {monitorcoord}

endsuperimp
```

Fig. 3. A superimposition for monitoring assignment statements.

executing, the remaining count is sent to the monitoring process, which will tabulate the results (indicated by the word "report").

Each instance of the roletype *participant* has a parameter indicating after how many assignments an update should be sent, while the added process, *monitorcoord*, has a parameter with the number of processes that will be instantiated with the roletype. By writing **include** *participant*(100), a process could send messages to *monitorcoord* every 100 assignments. Another process could send at dynamically changing intervals by using a local variable v that will, due to the internal logic of the basic computation, occasionally become equal to the counter, by stating **include** *participant*(v). After the global **use** *monitoring*, the line **include** *monitorcoord*(15) could appear to indicate that 15 basic processes will act as participants.

Note that if *assignsum* were to be returned to the basic algorithm so that it could subsequently use that information, the variable would have had to be given as a parameter. Also, for a more realistic situation where numerous assignments could be executed in each step, a variant without the restriction in the **global condition** clause could modify the *participant* roletype, using $count < c$ instead of $count \neq c$ in the transform of the guard, and $count \geq c$ instead of $count = c$ in the guard of the new rule. In that case, a message with the value of *count* would be sent when it was detected that at least c

assignments had been executed. In the situation described, it is simply inefficient to send *count* precisely when *c* is reached, even though this could be done at the cost of comparing the values after every assignment within a step, using transformations to express the superimposition.

3.2 Termination Detection

Turning to problems more specific to distributed programming, consider the superimposition declaration seen in Figure 4. This is a version of a termination detection algorithm due to Topor [24] (which in turn is based on several earlier algorithms, as explained in the reference). The idea of the superimposition is that a "wave" of colored tokens will travel from the leaves to the root of a spanning tree of the processes. If the basic algorithm has not terminated, this will be sensed in the root, and a *repeat* wave is sent to the leaves, so that the token wave can be reinitialized. The computational model is a distributed language with either synchronous *send* and *receive* operations, or at least with an assumption that a message will be sensed by the receiving process before an entire wave can traverse the spanning tree. This assumption is expressed in the superimposition declaration by the phrase "semisynchronous communication" in the **global condition** section. If it did not hold, the algorithm could be shown to be incorrect for messages that remain in nontree channels for an entire round of a token wave and a repeat wave.

The information on whether termination has occurred is encoded into the color of the tokens and a *node* variable from each process. Whenever a node sends a (basic) message, it will color its *node* variable black, in order to denote that it may have disturbed one of the nodes that was previously idle. An idle leaf sends a white token. An internal node will pass on a white token only if it has received a white token from all its sons and is itself white, and otherwise will send a black token after receiving tokens from all its sons. If the root is white and has received white tokens from all its sons, then termination has been detected. The correctness of the superimposition is proven in [24], although the crucial assumption on semisynchronous communication is expressed somewhat vaguely. In fact, the details of the justification are of little interest to the designer of a system interested in applying the superimposition.

The connection of the basic computation to the superimposition is through the formal parameters *idle* and *finished*, while the needed spanning tree is encoded into the parameters *parent* and *children*. The condition section expresses that if the variable that will be bound to *idle* is true, then the (augmented) basic algorithm in the associated process will only respond to new incoming messages. The (implicit) requirement that the variable *node* must be set to *black* whenever a message is sent (and before *idle* becomes true) can be achieved by adding the indicated assignment in the **transform** section immediately after every *send* statement. Thus, *node* can be local to the declaration. As may be seen in the result section of the source role type, the variable that is associated with the formal parameter *finished* in the *source* process indicates when termination has been detected.

```
superimp termination-detection
type color = (black, white);
global condition nodes in children form a spanning tree, c ∈ children(p) ⇒ p = parent(c),
        semi-synchronous communication

roletype source( idle:boolean; children:set of process; numbchild:integer; finished:boolean); exactly 1
var token, node, x :color; numbtokens:integer
result (finished ⇒ basic computation is done) and (basic computation is done ⇒ eventually finished)
condition idle ⇒ basic computation will occur only if a message is received
transform send <message> to <targetprocess> to * ; node := black
initialize node:= white; token:= white; numbtokens:= 0; finished:= false
rules
    [numbchild = numbtokens ∧ (¬idle ∨ node=black ∨ token=black) →
                                send repeat() to children;
                                node:= white; numbtokens:= 0; token:= white ]
    [receive tokenmsg(x) → if x=black then token:=black;
                           numbtokens:=numbtokens+1 ]
    [numbchild=numbtokens ∧ (idle ∧ node=white ∧ token=white) → finished:=true;
                                                                numbtokens:= 0 ]
endroletype source


roletype internal(idle:boolean; children:set of process; parent:process; numbchild:integer);
var token, node, x:color; numbtokens: integer;
condition idle ⇒ basic computation will occur only if a message is received
transform send <message> to <targetprocess> to * ; node := black
initialize node:= white; token:= white; numbtokens:= 0
rules
    [idle ∧ numbchild=numbtokens → if token=white then send tokenmsg(node) to parent
                                   else send tokenmsg(black) to parent;
                                node:=white; numbtokens:=0; token:=white ]
    [receive tokenmsg(x) → if x=black then token:=black;
                           numbtokens:=numbtokens+1 ]
    [receive repeat() → send repeat() to children]
endroletype internal


roletype leaf(idle:boolean; parent:process);
var token, node: color;
condition idle ⇒ basic computation will occur only if a message is received
transform send <message> to <targetprocess> to * ; node := black
initialize node:=white; token:=white
rules
    [idle ∧ token=white → send tokenmsg(node) to parent;
                          token:=black; node:=white]
    [receive repeat() → token:=white]
endroletype leaf

endsuperimp
```

Fig. 4 A superimposition for termination detection.

3.3 Bounding Monotonically Increasing Values

Even in somewhat more specific contexts, this type of modularity can be useful. One such example can be seen in [19], where an algorithm is

presented for preventing the formation of a cycle in a distributed directed graph with dynamic addition and deletion of edges. The algorithm uses monotonically increasing ranking values in order to prevent the formation of a cycle, but the values may become arbitrarily large. In order to bound the range of the rankings, a superimposition is used over the basic algorithm. The superimposition determines that a value outside the desired range is about to be assigned, adjusts other values to ensure that the ordering relation is maintained, and then assigns a value from within the permissible range. Superimpositions with similar goals of bounding counters or timestamps may be seen in [17] and [9].

The specification of such a superimposition should generalize the specific context so that it might be applied in other situations as well. Thus the applicability conditions only require that the collection of variables in the basic computation to be associated with the rank have a partial ordering relation at each moment (and the relation may vary according to the computations of the basic algorithm). However, the absolute values of those variables are not needed in the specification. The result assertion then guarantees that the partial ordering is maintained in the combination, and in addition the range of the variables is bounded by *bound*.

The first part of the superimposition declaration is:

```
superimp rankbound
  roletype any(val, bound: integer)
  var activate: boolean; t: integer
  transform
    val := ⟨expr⟩ to t := ⟨expr⟩;
                          if t > bound then activate := true else val := t
    ⟨guard⟩ to ¬activate; *
```

This means that the basic computation must be augmented, so that if in the original computation the rank variable corresponding to *val* exceeds the bound, then this is sensed in the superimposition, and the regular computation is temporarily disabled in that process so that appropriate steps can be taken by the code in the **rules** section of the declaration. The algorithm to achieve this is complex and is not presented here. Of course, it will include the resetting of *activate* to *false*, so that the basic computation may continue after the superimposition part has adjusted the rank values. There clearly will be assignments to *val*, thus changing values from the basic algorithm.

Note that in this example the superimposition modifies the values of variables from the basic algorithm. In the previous examples, the superimposition adds auxiliary variables and may modify the control flow of the basic algorithm, but only changes the values of basic boolean variables. In many examples, these can be assumed to affect only the control of the basic algorithm. Although sometimes this is defined to be a requirement of a superimposition, it seems too restrictive and does not cover many natural examples, including the one here. Thus, the more liberal requirement of not modifying the basic *specification* is adopted, as explained in the following section.

4. SPECIFICATION AND CORRECTNESS

A more precise statement of the assertions in a superimposition declaration requires a formalism for their expression. One possibility is to use a logic based on (past and future) temporal operators [22] and additional predicates about the state of the basic computation. In this formalism, the $\Box$ modal operator is interpreted as "from now on" (or "always" if it appears on the top level of the assertion), the $\Diamond$ modality is to be read as "eventually," while S is the "since" operator. Among the predicates we might have regarding the state are *assignments*(x), which is true when x is an assignment stateent, *executed*(s) true when a statement s has just been executed, or *terminated*(B) if no more statements of B can be executed in the possible executions from the present state.

An assertion that "b is true iff a statement t of the basic algorithm has executed since b was last false" could then be written as

$$\Box(b \equiv (executed(t)S(\neg b))).$$

An assertion that "the predicate $\neg b$ repeatedly becomes true" would be

$$\Box\Diamond\neg b.$$

The result assertion involving the variable *finished* from the *source* roletype in the termination detection superimposition is then

$$\Box((finished \Rightarrow terminated(basic)) \wedge (terminated(basic) \Rightarrow \Diamond finished)).$$

A superimposition is *correct* if, whenever processes from the basic algorithm are bound to roletypes with actual parameters so that (a) the restrictions on the bindings to the roletypes hold and (b) the global and local conditions are true for the appropriate actual parameters, then the assertions in the result section, with the actual parameters substituted, will be true of the superimposure. A frame axiom also holds, stating that the specification of the basic algorithm is true for the superimposure.

Note that it is an oversimplification to define the specification of the combination as the conjunction of whatever is done by the basic computation and the result assertions of the superimposition, since there may be contradictory assertions in the result of such a conjunction, which is not the intention. For example, in a termination-detection superimposition, the basic computation usually has the property that it may deadlock. On the other hand, since this is the purpose of the detection superimposition, the combination clearly does not deadlock and instead will properly terminate. Thus, it seems necessary to specify that the result assertions hold, as well as any property of the basic computation that is "not affected" by superimposition.

It is not always trivial to separate those properties superseded by the result assertions and those that continue to hold. Here it is assumed that any properties superseded were semantically true in the basic computation, but were not part of its *specification*. This is reasonable because those properties were presumably undesirable. Under such an assumption, the conjunction of the basic specification and the superimposition's result assertions will hold in the superimposure.

To summarize this view in a precise statement, the assumption is also made that a semantic interpretation is available both for programs and for specifications, and that both in fact define families of possible execution sequences. The following notation is used:

$\mathbf{B}$	a basic distributed algorithm
$\mathbf{P}$	a specification of **B** (including desired properties only, as noted above)
$(\mathbf{S}, \mathbf{C}, \mathbf{R})$	a superimposition, with **S**, the superimposition code; **C**, the applicability conditions and restrictions on roletype bindings; and **R**, the result assertions.
$\mathbf{A}^{\mathbf{x}}_{\mathbf{y}}$	the assertion or code **A**, with **y** substituted for **x** (where **x** and **y** may be tuples of the same arity).
$(\mathbf{B} + \mathbf{S}^{\mathbf{x}}_{\mathbf{y}})$	a superimposure of **S** on **B** with **y** the actual parameters of **B** corresponding to the formal parameters **x** of **S**. Note that this includes the transformations to the basic algorithm, the additional rules added by the superimposition, the initialization, and the finalization.
$(\mathbf{D} + \mathbf{S})$	a dummy superimposure on an arbitrary basic program **D**, with actual parameters identical to the formal ones in the declaration of **S**.
$\mathbf{X} \Rightarrow \mathbf{Y}$	semantic implication. The program or assertion **X** defines execution sequences that are a subset of the set of sequences defined by **Y**.

Using the notation above, a combination of a superimposition $(\mathbf{S}, \mathbf{C}, \mathbf{R})$ and a basic algorithm **B** with specification **P** is *correct* if whenever $\mathbf{B} \Rightarrow \mathbf{P}$, and $\mathbf{B} \Rightarrow \mathbf{C}^{\mathbf{x}}_{\mathbf{y}}$ (for variables **y** of **B** and parameters **x** of the superimposition), then

$$(\mathbf{B} + \mathbf{S}^{\mathbf{x}}_{\mathbf{y}}) \Rightarrow \mathbf{R}^{\mathbf{x}}_{\mathbf{y}} \wedge \mathbf{P}.$$

That is, if a basic algorithm satisfies both its specification and also the conditions for applying a superimposition when **y** is substituted for **x**, then the superimposure with that binding satisfies both the result assertion with that substitution and the specification of the basic algorithm.

To show that a superimposition declaration is correct, independently of a specific basic algorithm, prove that if both **C** and $(\mathbf{D} \Rightarrow \mathbf{C})$ are assumed, then

$$(\mathbf{D} + \mathbf{S}) \Rightarrow \mathbf{R}$$

holds. The assertion **C** contains all relevant information about the basic algorithms to which the superimposition may be applied, and **D** is a dummy basic program about which nothing is known except that **C** can be assumed true. Note that any specific basic algorithm can never change variables from the superimposition (due to the scoping rules), and that the condition **C** must be shown to hold for any basic algorithm to which a superimposition is to be applied. Therefore, if the above semantic implication holds for a superimposition, any possible combination will satisfy the superimposition result assertion.

For the examples from the literature, the assertion above is demonstrated through either informal or formal program-correctness techniques. Thus, in the termination detection example (3.2), the proof of [24] shows that the specification holds for any basic program and configuration of processes that obey the restriction on the variable *idle* (that no spontaneous messages are sent while *idle* is true) and for which a spanning tree is given.

It remains to show that, in the combination, the basic specification is not violated. If the superimposition does not change variables that are formal parameters, and none of the statements of the basic algorithm are removed or modified (except to add assignments to superimposition variables), it follows from the definitions that the computations of the basic algorithm are unchanged in the superimposure, and thus they continue to satisfy the specification as previously. Of course, it can be determined syntactically whether a superimposition satisfies these conditions. Otherwise, in the case of more general transformations, for each superimposure (i.e., set of bindings seen in the **include** and **use** statements), it must be shown that the basic specification is indeed unchanged in the superimposure, again using verification techniques.

In order to show this generically, the condition **C** must include the properties of the basic program that guarantee the invariance of the basic specification under the augmentation due to the superimposition. Then assuming **C**, and $\mathbf{D} \Rightarrow (\mathbf{C} \wedge \mathbf{P})$, the semantic implication

$$(\mathbf{D} + \mathbf{S}) \Rightarrow \mathbf{P}$$

should be true. That is, if the dummy basic program is assumed to satisfy **C** and some specification **P**, the combination must still satisfy **P**. Thus, for examples of the type seen in Example 3.3, **C** includes the fact that only the relative order of the rank or timestamp is significant for the correctness of the basic algorithm, and not the precise values. The superimposition may only be applied to basic algorithms with that property.

It would clearly be desirable to identify an intermediate level of "acceptable" transformations that exclude a complete rewriting of the basic algorithm. This is discussed in the following section.

5. VARIETIES OF SUPERIMPOSITIONS

As mentioned in the previous section, one way to guarantee that the basic algorithm is unaffected by the superimposition is to forbid modifying the execution sequences of the basic algorithm in any way, as in [6]. Thus, since a monitoring superimposition (e.g., Example 3.1) and the basic computation are presumably unrelated, in this case the result assertion guarantees all of the specification of the basic computation, plus assertions about the values in the statistical summary (e.g., that a variable *assignsum* represents the number of assignments executed in all the processes). However, this seems too restrictive, especially outside of the (global state) Unity context. The only possible superimpositions are those to monitor or detect properties, without

any possibility of changing the computations so that the basic variables satisfy additional properties.

A more liberal approach is taken in [3], where a superimposition can only affect the *control* of the basic algorithm. For example, guards could be made untrue, thereby closing previously available control paths. If the variables that correspond to the parameters *idle* and *finished* could be shown not to affect the "real" variables of the basic computation, and only to close some of the guards, this restriction would hold for Example 3.2. The intention of this restriction is to ensure that all safety properties of the basic program continue to hold automatically in any superimposure, although liveness properties could be changed. If the basic program of each process has the form of a single loop, in the normal form assumed in most of the paper, this is indeed true. However, if a process may have any code following the first loop, even such a limited superimposition could cause proper termination of the first loop where previously it deadlocked. In that case, assertions that were invariants for the basic program could now be invalid when the code after the loop is reached and executed. Thus, even safety properties cannot be guaranteed to still hold if the form of the program is slightly more general.

Moreover, such a view is inadequate for examples where the values of basic variables *may* be changed, without disturbing the specification. Example 3.3 and other algorithms to bound counters or timestamps have all of the characteristics of a superimposition: they act in conjunction with another basic distributed algorithm over the same state, add the property of keeping the counters within a bounded domain without violating the specification of the other algorithm, and they are described uniformly, independently of the basic algorithm. Yet they only make sense if the counters or timestamps of the basic program can sometimes be modified so as to guarantee remaining in the bounded range of values. Other examples naturally included in the category of superimpositions are algorithms to reorganize a data structure of the basic algorithm in order to increase efficiency, without affecting the abstract data operations of the basic algorithm.

If it is desired that later stages of the basic algorithm take some action based on the result of the superimposition, the approach of not affecting the basic variables is insufficiently expressive. For example, when deadlock has been detected, the superimposition could include a breaking of the deadlock by sending messages to be treated by the basic code, or by changing the values of variables that indicate a passive local state. An approach that allows closing control paths (guards) can be used to force proper termination on a program that otherwise would have the processes indefinitely waiting for messages that will never arrive, but cannot be used to break a general deadlock where a (minor) correction and continuation is desired.

As another example, a *reliability* superimposition could guard against message loss by requiring that an *acknowledge* message be received in the process sending a message, before it continues to its next instruction. However, this activity is only reasonable if, occasionally, an acknowledgment is not received within a period known as the time-out. In this case, the basic

message must be resent. But the other approaches have no way to indicate this to the basic algorithm, or to have the basic algorithm receive and treat a message sent by the superimposed part. Information may be transmitted back to the basic algorithm only by changing basic variables.

Such transformations should at least capture many of the modifications that seem intuitively natural and appear in examples, informally called superimpositions. The quality and elegance of a superimposition is inversely proportional to the complexity of the interface with the basic algorithm, and in particular to how much is done by the **transform** section.

Besides the addition of parametrization, which other definitions of superimposition do not treat, there is a difference in the assumption about the degree of similarity of the configuration of basic and superimposed code. In particular, here new processes can be added by the superimposition, as well as new communication channels. That is, two augmented processes in the superimposure may communicate (through the superimposed code) even though the corresponding basic processes did not. If it is required that no new channels be added by the superimposition (as is assumed in [3]), this may be given as a restriction on the possible processes of each roletype.

6. CONCLUSIONS

A control structure has been presented to allow convenient expression of the superimposition of one algorithm on another. Although an interleaving at the level of guarded commands or rules has been assumed, the ideas are orthogonal to the degree of interleaving. In general, superimposition can be adapted to any distributed model, much like the *script* mechanism seen in [14] serves as a general construct for communication-closed layers.

For most distributed languages, the view seen here of augmenting the basic processes is appropriate. Another convenient framework for adding a slight variant of superimposition may be seen in the Guardian construct of Argus [23], where processes with shared memory can be combined into a guardian. In that context, instead of interleaving rules of the superimposition with the basic code, a roletype of the superimposition would be adjoined to each process of the basic computation in the same guardian. The added roletype can examine the common memory and execute the superimposed code in parallel to the basic computation. In this case the roletypes involved in the superimposition remain distinct from those of the basic computation, even though they are distributed among the guardians.

It is still somewhat unclear when a decomposition to a basic computation and a superimposition is desirable or possible. The difficulty is similar to that of any other method of decomposition. One approach would be to develop a library of potential superimpositions, which are applicable to large classes of basic computations. At least in those cases it is clear that an algorithm that is only loosely linked to the basic computation can be found for the additional functionality guaranteed by the superimposition. This is the natural way to treat general superimposition algorithms found in the literature, since they have wide applicability. However, it should be noted that the concept can also

be useful for the decomposition of a specific task to relatively independent subtasks, even when the superimposition part is not of general applicability.

ACKNOWLEDGMENTS

Mike Evangelist, Ira Forman, and Nissim Francez have provided valuable comments on the ideas presented here.

REFERENCES

1. APT. K. R., AND CLERMONT, PH. Two normal form theorems for CSP programs. RC 10975, IBM, T. J. Watson Research Center, Yorktown Heights, N.Y., Feb. 1985.
2. BACK, R. J. R., AND KURKI-SUONIO, R. Decentralization of process nets with centralized control. *Dist. Comput., 3*, Springer-Verlag, Berlin, Germany (1989), 73–87.
3. BOUGE, L., AND FRANCEZ, N. A compositional approach to superimposition. In *Proceedings of ACM POPL88 Symposium* (Jan. 1988).
4. CHANDY, K. M., AND LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst., 3*, 1 (Feb. 1985), 63–75.
5. CHANDY, K. M., AND MISRA, J. An example of stepwise refinement of distributed algorithms: Quiescence detection. *ACM 8*, 3 (July 1986), 326–343.
6. CHANDY, K. M., AND MISRA, J. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Mass., 1988.
7. CHANDY, K. M., MISRA, J., AND HAAS, L. Distributed deadlock detection. *ACM Trans. Comput. Syst., 1*, 2, (May 1983), 144–156.
8. DIJKSTRA, E. W., AND SCHOLTEN, C. S. Termination detection for diffusing computations. *Inf. Process. Lett., 11*, 1, North-Holland, New York (Aug. 1980), 1–4.
9. DOLEV, D., AND SHAVIT, N. Bounded concurrent time-stamp systems are constructible. *Proceedings of 21st ACM Symposium on Theory of Computing* (May 1989), 454–466.
10. DONAHUE, J. E. Complementary definitions of programming language semantics. *LNCS 42*, Springer-Verlag, New York, 1976.
11. ELRAD, T., AND FRANCEZ, N. Decomposition of distributed programs into communication closed layers. *Sci. Comput. Program. 2*, 2 (1982), 155–173.
12. FORMAN, I. On the design of large distributed systems. MCC Tech. Rep. STP–098–86 (rev. 1.0), Jan. 1987. Preliminary version in *Proceedings of International Conference on Computer Languages*, (Miami Beach, Fla., Oct. 1986).
13. FRANCEZ, N. Distributed termination. *ACM Trans. Program. Lang. Syst., 2*, 1 (Jan. 1980), 42–55.
14. FRANCEZ, N., HALPERN, B., AND TAUBENFELD, G. Script: A communication abstraction mechanism and its verification. *Sci. Comput. Program., 6* (1986), 35–88.
15. GERTH, R., AND SHRIRA, L. On proving communication closedness of distributed layers. In *Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science* (New Delhi, 1986), *LNCS 241*, K. V. Nori, Ed., 330–343.
16. HAREL, D., AND PNUELI, A. On the development of reactive systems. In *Proceedings of Advanced Institute on Logics and Models for Verification and Specification of Concurrent Systems* (La Colle Sur Loupe, Oct. 1984).
17. ISRAELI, A., AND LI, M. Bounded time stamps. In *Proceedings of 18th IEEE FOCS Symposium*, 1987, 371–382.
18. KATZ, S., AND PELED, D. Interleaving set temporal logic. *Theor. Comput. Sci., 75* (1990), 263–287.
19. KATZ, S., AND SHMUELI, O. Cooperative distributed algorithms for dynamic cycle prevention. *IEEE Trans. Softw. Eng. SE-13*, 5 (May 1987), 540–552.
20. LAMPORT, L. Solved problems, unsolved problems, and non-problems in concurrency. In *Proceedings of 3rd ACM PODC Symposium* (Vancouver, 1984), 1–11.
21. LEHMANN, D., PNUELI, A., AND STAVI, J. Impartiality, justice, fairness: The ethics of concurrent termination. In *The Proceedings of 8th ICALP* (Acco, Israel, July 1981). *LNCS 115*, O. Kariv and S. Even, Eds., Springer-Verlag, 1981, 264–277.

22. LICHTENSTEIN, O., PNUELI, A., AND ZUCK, L. The glory of the past. In *Logics of Programs Symposium, LNCS 193*, Springer-Verlag, (New York, 1985), 196–218.
23. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst., 5*, 3 (July 1983).
24. TOPOR, R. Termination detection for distributed computations. *Inf. Process. Lett., 18* (Jan. 1984), 33–36.

Received November 1987; revised November 1989, April 1992; accepted July 1992