Towards Languages that Support Program Derivation, or Control Modularity Considered Harmful*

REINO KURKI-SUONIO

Tampere University of Technology
Box 527, SF-33101 Tampere, Finland
e-mail: rks@tut.fi ,

Abstract

Of current trends in programming languages, the paper concentrates on the need to support formal specification and derivation of software, mainly in the context of reactive systems that are in continual interaction with their environments. The non-programming facilities of operational specifications are briefly analyzed, and their inclusion in design oriented specification languages is considered. Early commitment to control-oriented decisions is found harmful, which leads to a language basis with implicit concurrency and no notion of control flow. The advantages of this approach for certain intuitively natural methods of program derivation are demonstrated. The paper ends with general comments about the diversification of languages along the dimension of specification, design, prototyping, and implementation.

1. Introduction

Software objects are artifacts that cannot be classified either as concrete objects or as pure abstractions. An executable machine language program might be considered a concrete object with the original source code as its abstraction. However, source programs are also executable, at least in principle, and therefore equally concrete. On the other hand, no matter which level of languages is considered, each program is an abstraction of something that gets concrete only in its physical execution. Therefore, as pointed out by Lamport [La89], every program is a specification, and some specifications are implementations of other specifications.

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

The evolution of programming languages shows continual rise in the *level of abstraction,* which means that the specification nature of programs becomes more and more obvious. The programmer needs to worry less about "concrete" or "efficient" representations, and can concentrate more on "abstractions" that are easier to reason about and can be automatically compiled into lower levels. Some perspective on this trend is provided by the remark by Parnas [Pa85] that the term "automatic programming" was probably used for the first time in the 1940s in a paper by Saul Gorn on the possibilities of building a simple assembler.

Another direction, which is evident especially in the current development of so-called object-oriented languages [SW87], is towards structures that facilitate *prototyping,* easy modification, and evolutionary development of systems. It is not entirely clear, however, whether the resulting language flexibility can be adequately reconciled with the security and provability requirements of many applications.

Thirdly, only very small programs can be written directly, and software maintenance involves changes in their specifications. The increasing significance of program reliability and correctness therefore requires explicit support for *software derivation from specifications.* Although most abstractions in programming languages, such as block structure, subroutines, and modules, support various methodologies for programming-in-the-small, explicit support for program derivation has usually not been considered a programming language issue. Instead, extra-linguistic tools have been provided in operating systems and programming environments for combining pieces of software, as well as for version control and related aspects of software maintenance. Even though some languages like Lisp and Smalltalk come with integrated programming environments, and Ada was claimed to extend the scope of programming languages towards programming methodologies and utilization of program libraries, only elementary language support is presently available for the derivation of software.

Any substantial support for program derivation requires the use of *formal specifications* instead of the informal and semiformal approaches that still dominate in programming practice. Like programs, formal specifications cannot be given directly, and their derivation is similar to that of programs. Existing components are used in this process, new properties are introduced in a stepwise manner, the level of abstraction is lowered for reasons like efficient implementability, and the results need to be verified and validated. Attempts to support software derivation have lead to experimental *wide spectrum languages* [Ba&89] within which specifications can be transformed into implementations through correctness-preserving transformations.

Notice that it is not only the result of the derivation process that is important, since the higher levels provide important abstractions and insight that are lacking in the final (high-level language) form. The development of tools to analyze finished designs, in order to recover the insight that was never made explicit in the first place, is a backward approach, and is no substitute for the derivation of programs in a manageable way from higher-level specifications. In particular this is currently a problem with concurrent systems, where most so-called specification languages have no better abstractions of concurrency than those available in programming languages.

The above trends in programming languages provide the background for this paper. We shall mainly focus on *reactive computations* [Pn86], in which the system

is in continual interaction with its environment. Obviously, traditional input-output computations are a special case of these.

The structure of the paper is as follows. First we address the question of what distinguishes some specifications from programs. In Section 3 the problems of concurrency — or, rather, independence of sequentiality — lead us to a somewhat radical conclusion about the usefulness of traditional control flow oriented modularity. When the notion of control flow is abandoned, *statecharts* [Ha87] are shown to be suitable for the structuring of the global state. In Section 4 we demonstrate how independence of control decisions leads to a language basis that is suited for certain intuitively natural methods of program derivation. The paper ends with some general comments about the diversification of languages along the dimension of specification, design, prototyping, and implementation.

The paper is heavily influenced by and biased towards *joint action systems*, developed together with Ralph Back [BK83, BK88a, BK88b]. Case studies of such systems and design methods for their development have been investigated in [BK83, BK84, Ku86, KK88, Ku89], and [KJ89, Jä&89] report on an experimental specification language DisCo (for *Dis*tributed *Co*operation) that is based on these ideas. The reader is also reminded of the close similarity between joint action systems and the Unity language by K. Mani Chandy and Jay Misra [CM88a].

2. Operational Specifications and Programs

Formal specifications are commonly understood to express *safety* and *liveness* properties of programs [AS85]. Informally, the former state that nothing bad will ever take place, while the latter express the requirement that the desired good things will eventually happen. Temporal logics are well-established formalisms in which such properties can be expressed [MP83, Pn86]. Obviously, such specifications do not cover all formalizable requirements: statistical efficiency properties, for instance, remain inexpressible.

Programs are operational and executable specifications. In general, an operational specification consists of two components: a *generative* mechanism that is based on computational steps, and a set of *constraints* [Fe87]. The former generates a collection of potentially possible (finite or infinite) computations, which is then restricted to a subset by the constraints. Notice that operationality is more a viewpoint than a well-defined property. Temporal logic specifications, for instance, can be understood in these terms by viewing the temporal properties as constraints on the implicitly generated collection of all possible sequences of events.

In programs the emphasis is on the generative mechanism, which determines safety properties only. Liveness properties are given by implicit constraints that exclude those finite computations that have not yet terminated, as well as such infinite computations that do not satisfy certain *fairness* requirements [Fr86]. Since we are interested in programs as abstract specifications, we ignore here the practical non-constructivity and non-verifiability issues of fairness constraints [Di88, SL88, CM88b].

Since each program is a specification, and all specifications have an operational interpretation, the question arises whether there is any fundamental distinction

between specifications and programs. Is the difference only in efficiency, whose significance changes rapidly with the development of hardware and software technology?

Based on Dijkstra's weakest precondition calculus of predicate transformers [Di76], Ralph Back was the first to introduce a mixed formalism in which specifications and programs coexist on equal basis [Ba78, Ba80, Ba88a]. For specifications, one of Dijkstra's healthiness conditions for programs had to be relaxed. The reason was in the need for *unbounded nondeterminism.* By this we understand the selection of a value that satisfies a given condition, when the number of potential alternatives is infinite. If several alternatives satisfy the condition, then the choice between them is nondeterministic.

Unbounded nondeterminism is equally non-constructive as fairness. Notice that describing the input of an arbitrary integer as a single event leads to unbounded nondeterminism, while replacing this by a sequence of separate input events for an arbitrary number of digits requires fairness instead.

Similarly to the bounded nondeterminism in [Di76] this nondeterminism is of the *demonic* variety, which means that each possible choice must lead to a correct computation.

Recently Back and von Wright [BW89] have extended this work to remove also the other healthiness conditions, except for monotonicity. The results are mathematically appealing, and they can be interpreted to lead to two further possibilities in specification languages: *angelic nondeterminism*, which only requires that at least one of the alternative choices leads to a correct computation, and *miracles,* which miraculously succeed in establishing conditions by impossible assignments. Similar generalizations have been found desirable also elsewhere [dB80, Mo87, Ne87, Mo88a, Mo88b], and their need for describing practical specification languages has been observed [Mo88b, Ba88b].

From a more practical viewpoint, at least the following quasi-executable facilities have been found useful in operational specifications:

- The generative mechanism may involve *unbounded nondeterminism* [BB87, KJ89]. Notice that unbounded nondeterminism is implicitly present in specifications that do not use an explicit generative mechanism, as is the case with temporal logic [Pn86] and algebraic specifications [Ba89].
- A computation may refer to its *past history* without having explicitly recorded it [BG79, Fe87, AL88].
- There can be "*oops conditions*" that are not allowed to become true [BG79, Fe87]. If the generative mechanism leads to such situations, the constraints are assumed to exclude those computations.
- A computation may contain *prophetic references* to its future [BG79, Fe87, AL88]. The constraints are then assumed to exclude computations where such predictions would turn out to be incorrect.

Of these facilities, references to past history are the least problematic for direct implementation, as further recording of history can always be added. Unbounded nondeterminism also looks rather innocent. Existential quantification provides, however, extremely powerful possibilities for implicit solutions of problems for which no algorithms are known.

In the absence of unbounded nondeterminism, oops conditions cause no obstacles for classical input-output computations, and backtracking is a standard technique for their implementation. With unbounded nondeterminism this need not succeed, however, and with reactive computations the situation becomes totally different, as there is no way for an implementation to withdraw interactions that have already taken place between the system and its environment.

The situation with prophetic references is no simpler, as they can be understood as nondeterministic guesses about the future, combined with oops conditions to be evaluated later.

From this discussion we can conclude that all specifications are not programs, even if efficiency considerations are totally ignored, and that the differences are even more significant for reactive systems. A good specification language therefore needs facilities that do not satisfy the constructivity criteria for programming. In view of the problems in removing their use by systematic transformations [LF82] one should, however, be cautious in introducing them in design-oriented specification languages. Some form of unbounded nondeterminism seems to be a minimum that is required by reasonable design methodologies [Ku89].

3. On Concurrency, Control Modularity, and Structure of Global State

Concurrency is often thought of as an auxiliary feature that can be added afterwards to any description language. Its use seems to complicate matters, and one may therefore try to avoid it as far as possible. Even when concurrency is crucial for the design, one may resort to the backward approach of first designing a sequential solution and then parallelizing this. Notice that for the description of reactive systems concurrency is always essential, even when the system is to be implemented as a single process, since the environment works concurrently with the system itself.

Our view of concurrency is different: to us sequentiality or any particular choice of parallelism is an implementation-oriented design decision, of which specifications should be independent. We therefore argue that good support for deriving software from specifications cannot be provided by amending a sequential base language with additional constructs for concurrency. In fact, instead of having specific constructs for sequential and concurrent control, the base language should be independent of any such choices. In other words, it is not concurrency in itself that is important, but *independence of control decisions.*

In mathematics it is often the case that a more general formulation makes a problem easier to manage. The same has also been observed in programming. For instance, the advantages of nondeterminism over strictly deterministic descriptions were clearly demonstrated by Dijkstra [Di76], even in situations where the programmer would later restrict the design by purely deterministic choices. The situation with concurrency is similar, and we claim that it is the conventional control-oriented modularity that has prevented us from realizing this.

Any execution model of computing involves a *state* (memory, registers, variables) and *actions* (instructions, statements, transitions) that modify this state. Conventionally the state is partitioned into a *data* part (accumulators, variables) and *control* part (instruction counters, control states). In the light of sequential programming

and von Neumann computer architecture this approach is natural, and the control-oriented modularity of structured programming may therefore seem inherent to any well-structured description of computations. It is this assumption that we challenge, and in doing so we need to abandon the early partitioning of state into data and control parts. For further discussion on why control modularity is especially harmful in the design of parallel programs, the reader is referred to Chandy and Misra [CM88a].

Without the special role of control state the conventional control flow oriented modularity becomes inapplicable. Even though independence of control decisions can easily be achieved, the result may be chaotic. Similar ideas have been used (with different motivation) in production system languages like OPS5 [FD77], and the drawbacks are well-known. With no notion of control flow there is no built-in structure in the collection of actions, and one is easily led to encode the missing control flow in unstructured collections of bits and flags. Obviously, such an unmodular system is even more difficult to understand than one where the control state has been explicitly separated from data.

In the following our purpose is to show that abandoning control flow does not imply lack of structuring and modularity. On the contrary, our conclusion will be that this can lead to another kind of modularity that is especially suited for software derivation. We start by inspecting how to impose structure on a state that has no dedicated control components.

We assume that the global state of a system is partitioned into components called *objects*. The state of a single object is called its *local* state. From an implementation point of view objects may be thought of as either data structures or processes. Avoiding the notion of control flow implies that no distinction is made between passive objects (data structures) and active agents (processes). Therefore, objects require structuring capabilities that are equally suited for both.

Harel's *statecharts* [Ha87] turn out to be an ideal visual formalism for this purpose. From the viewpoint of active agents, their hierarchical state structure generalizes the notion of ordinary finite-state systems, and can be interpreted as the nested control structures in conventional high-level languages. Associating data items with the states makes this analogy even more complete. On the other hand, from the viewpoint of passive objects, statecharts can be interpreted as record structures containing tagged unions of alternatives. The state transitions of a statechart correspond to the actions of the system, which are now separated from the structure of the global state.

As a simple example, Figure 1 gives a statechart description of database clients. On the outermost level a client object has three exclusive states: *idle, starting* a transaction, and *engaged* in one. When *engaged,* a client is either *ready* to issue another request or *waiting* for a response. A *waiting* client is expecting a response either to an end request *(ending)* or to a read or write request *(accessing)*. Transitions of the statechart are labeled by identifiers in italics, referring to actions whose descriptions have been omitted. For simplicity, the data items that are associated with the states have also been left out.

This simple example illustrates *or* decomposition of states, in which case the immediate substates of a state are exclusive alternatives to each other. Another useful possibility is *and* decomposition, which means that the actual state is a Cartesian product of states in each component.

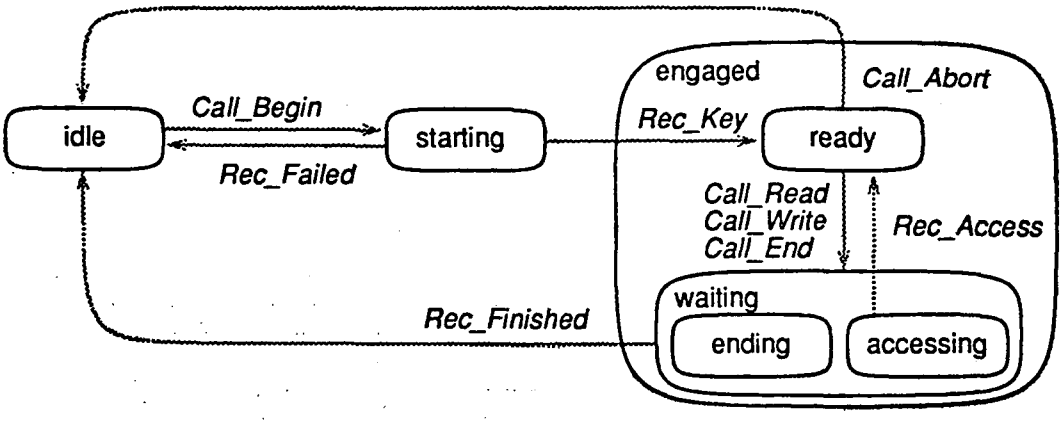

Figure 1. Statechart description of database clients

In the DisCo language the same information (except for the transitions) would be given in textual form as the following class declaration:

```
class Client is
    state idle, starting, engaged;
    extend engaged by
        state ready, waiting;
        extend waiting by
            state accessing, ending;
        end waiting;
    end engaged;
end Client;
```

The keyword **extend** is used to emphasize the possibility to abstract away internal structure of states, and to extend them later with further detail.

When the global state of a system is partitioned into objects with local states, the description of actions needs to be separated from objects. As there is no control flow to *enable* an action that can be executed next, each action requires a *guard* expression to determine its enabledness. Since any number of actions can be enabled at the same time (even though only one is selected for execution), nondeterminism is inherent in this kind of systems.

In principle, any number of objects may participate in an action in the sense that their local states are required and possibly updated in its execution. From the viewpoint of active agents the execution of an action can be interpreted as follows: first the participants determine by mutual communication that the action is enabled and perform a joint handshake to become committed to its execution; while committed to the action they exchange the data that are necessary for each participant to update its own local state appropriately; after updating its own local state and providing the other participants with the data they require, each object becomes free for another action. On the other hand, from the viewpoint of passive objects

we can think in terms of a centralized scheduler that evaluates guards and triggers the execution of enabled actions. It is important that both views are equally possible, i.e., that the base language has no explicit constructs for concurrency or communication, and is therefore independent of any control decisions.

4. Support for Program Derivation

The main topic of this paper is the development of language support for program derivation. So far we have described a language basis that is independent of control decisions; in this section we shall demonstrate how such a base language is suited for such useful methods of program derivation that would be much more complicated to express with conventional control flow oriented languages.

The main design method to be considered here is superimposition or *superposition*. This is a layered approach where, starting from an initial solution that satisfies some basic requirements, further properties are imposed without violating those already established. Superposition has been mainly used in connection with distributed systems, and different formulations of it have slightly different properties. An early use of the technique and the term was in [DS80]; in [LS84] it was described as the reverse of a protocol verification method; recently it has been suggested as a control structure for concurrent and distributed programming [Ka87, BF88]; in [CM88a] it was introduced as one of the main facilities for designing Unity programs in a modular fashion; in connection with joint action systems the technique has been applied in [BK83, BK84, Ku86, Ku89].

Here we introduce the method by a simple example that has sometimes been used in comparing different specification and design methods. The problem is to describe a doctors' office, which involves patients that are cured by doctors, and a receptionist that organizes the free doctors to treat the waiting patients.

We start with the simplest possible projection of the system that exhibits complete behavior by itself. In this case such a system contains only one kind of objects, patients, with two possible states, *well* and *sick*, and two kinds of actions: each patient that is *well* may become *sick*, and a *sick* patient may become *well*. In DisCo this could be described as follows:

```
system Patients is
    class patient is
        state well, sick;
    end;
    action get_sick by p: patient is
    when p.well do
        → p.sick;
    end;
    action get_well by p: patient is
    when p.sick do
        → p.well;
    end;
end Patients;
```

Each of the two actions has just one participant, a patient *p*, and a guard indicating that *p* is *well* or *sick*, respectively; the effect of the actions is to change the state of *p* as indicated by the state transition statements (→). Only the class declaration for the patient objects is given here; the actual creation of patient objects is assumed to be given separately in system initialization.

This first approximation of the system is easy to understand: patients just get sick and are cured nondeterministically. (For simplicity we omit here fairness questions like whether each *sick* patient is eventually cured.) Another layer is now superposed on this system, introducing the property that no patient is cured without a doctor. Each doctor has two states: *free*, or *busy* with a patient *p*. The state *well* of patients needs to be extended with two substates at this stage, to distinguish whether a cured patient has already checked out from the office or not. Action *get_well* is also refined to indicate the need of a doctor in this action, and a third action is introduced for releasing a cured patient from the office:

```
system Doctors with Patients is

    class doctor is
        state free, busy(p: patient);
    end;

    extend patient.well by
        state released, hide*checking_out;
    end;

    refined get_well by ... d: doctor is
    when ... d.free do
        → d.busy(p);
        ...
    end;

    action release by p: patient; d: doctor is
    when d.busy.p=p ∧ p.well.checking_out do
        → d.free;
        → p.well.released;
    end;

end Doctors;
```

Ellipses (...) belong to the language and indicate parts taken directly from the previous level. The refinement of *get_well* introduces an additional participant *d* and another conjuct to the guard, indicating that *d* must be free, and makes *d* become busy with patient *p*. The state *patient.well* is extended in such a way that a cured patient always enters the default substate *p.well.checking_out* (indicated by the star). This substate is hidden (**hide**) from the previous level in the sense that *get_sick* cannot be enabled in it. Therefore *p* has to participate in the new action release before getting sick again, i.e., a cured patient cannot get ill before leaving the office.

Provided that some number of doctors are initially created, the system is again complete, although it still lacks some of the required properties. In the next step we introduce a receptionist that organizes the free doctors and the waiting patients. Again, the creation of the initial state is omitted:

```
system Office with Doctors is

    class receptionist is
        pq: sequence patient;
        dq: sequence doctor;
    end;

    refined get_sick by ... r: receptionist is
    when ... do
        ...
        r.pq:=r.pq & p;
    end;

    refined get_well by ... r: receptionist is
    when ... p=first(r.pq) ∧ d=first(r.dq) do
        r.pq:=tail(r.pq);
        r.dq:=tail(r.dq);
        ...
    end;

    refined release by ... r: receptionist is
    when ... do
        r.dq:=r.dq & d;
        ...
    end;

end Office;
```

Notice that although the receptionist is needed in all actions, it does not create a bottleneck for a concurrent implementation. In *get_well*, for instance, the role of the receptionist is only to remove the doctor and the patient from their respective queues, after which it can start participation in some other action, while the doctor and the patient still continue in action *get_well*.

This example gives rise to the following general observations about superposition:

- It is a top-down design method in which even partially specified systems are given as complete systems exhibiting well-defined behavior.
- The global state of a system can be extended by adding new objects and by extending the local states of old ones. Statechart structuring of objects is especially suited for the addition of new substructures and new data components.
- New functionality can be added and new properties can be introduced by providing new actions and by refining the old ones. Atomicity of actions and absence of control flow for individual objects are significant for doing this smoothly. Additions and refinements are restricted to ones that do not affect the old state components.
- Nondeterminism of the system can be restricted by strengthening the guards of actions. For instance, the design may utilize unbounded nondeterminism until a basis for deterministic selections has been superposed.
- With the notions of objects and actions, all modifications have good locality: one logical change does not lead to several small changes in different places.

- The preservation of all safety properties can be guaranteed by language rules; the restrictions enforced are similar to what has been called complete compatibility in connection with object-oriented programming [WZ88]. Because of guard strengthening and the potential possibility for takeover by new actions, liveness properties have to be checked.

By this we hope to have demonstrated that, once the self-evidence of control flow oriented modularity is given up, it is possible to support effectively such intuitively natural approaches to structured derivation of programs that are quite complicated to manage with conventional language bases. In an ordinary multi-process program, for instance, a simple modification of a single action would correspond to changes scattered in the codes of all processes involved.

For brevity we have described here only one design method, superposition, which is based on a top-down approach. Similar observations concern, however, the bottom-up design method that is dual to superposition in the light of the above notions. This method introduces modularity with *communication-closed layers* [EF 82]. In our language it uses a mechanism called *inheritance* [KJ 89] and is especially suited for the development and utilization of reusable modules. As described in [Jä & 89], the mechanisms for supporting these two design methods can be understood as two well-structured variants of object-oriented inheritance. In other words, these ideas can also be described as an object-oriented approach to specification. Notice, however, the fundamental departure from conventional object-oriented programming that objects are not assumed to have individualistic behavior; the methods of individual objects are replaced by roles in cooperative, multi-object actions.

5. Concluding Remarks

In this paper we have investigated some novel language directions to which we may be led by the need to support program specification and derivation effectively, especially in the context of reactive systems. In particular, we hope to have demonstrated that early commitment to decisions on control is harmful for certain natural approaches to software derivation. Therefore, languages with a possibility for independence of control decisions are foreseen, and some capabilities of a simple experimental specification language of this flavor have been presented.

More generally, with this direction of language development the practical significance of the following views are expected to be emphasized:

- The apparent need of better tools for program analysis is an indication of inadequate languages; ultimately the only way to reliable programs is by formal specifications with proper abstractions and by well-structured derivation of programs from them.
- In providing effective support for program derivation it is insufficient to restrict to constructive programming facilities; programs have to be considered as special cases of more general constructions.

In order to cope with different language requirements for program specification, derivation, prototyping, implementation, etc., a wide spectrum language would need

a huge arsenal of capabilities. Various current trends in language development have different emphases in this respect, and we do not believe in the creation of languages that are very large along this axis. In spite of its size and ambitious objectives, Ada, for instance, extended the scope of programming languages only modestly towards supporting program development. Integrating effective support for program derivation with all the facilities of an efficient implementation language would necessarily lead to a language with even much greater complexity. To us this seems a hopeless direction, but, deciding from [Ga 89], the idea of such language dinosaurs has not been abandoned.

With a collection of different and more specialized languages the role of conventional high-level languages would change, which would also affect their requirements. Program specification and the initial design transformations could be carried out in languages with only little support for efficient executability, which was the area of the technical contributions in this paper. High-level languages that can be automatically compiled into efficient machine code would be needed as target languages for such design systems, and also as languages for efficiency-oriented transformations. However, the design motivations of current high-level languages have been quite different, and it would be instructive to evaluate them in the light of theses new uses.

Acknowledgments

The joint action approach was developed together with Ralph Back from Abo Akademi. Its use as a basis for a specification language, and the development of associated tools and design methodologies are the topics of project DisCo at Tampere University of Technology. This project is part of the FINSOFT programme of the Technology Development Centre of Finland (TEKES), and is supported by four industrial partners. Fruitful discussions with other project members and with a seminar group are gratefully acknowledged.

References

[AL 88] M. ABADI and L. LAMPORT, The existence of refinement mappings. Res. Rep. 29, Digital Equipment Corporation, Systems Research Center, Aug. 1988.

[AS 85] B. ALPERN and F. B. SCHNEIDER, Defining liveness. *Information Processing Letters 21*, (Oct. 1985), 181—185.

[Ba 78] R. J. R. BACK, On the correctness of refinement steps in program development. Report A—1978—4, Department of Computer Science, University of Helsinki, 1978.

[Ba 80] R. J. R. BACK, Correctness preserving program refinements: proof theory and applications. Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam 1980.

[Ba 88a] R. J. R. BACK, A calculus of refinements for program derivations. *Acta Informatica 25*, 6 (1988), 593—624.

[Ba 88b] R. J. R. BACK, Refining atomicity in parallel algorithms. Report A 57, Department of Computer Science, Åbo Akademi, 1988. To appear in *Conference on Parallel Architectures and Languages Europe*, 1989.

[BK 83] R. J. R. BACK and R. KURKI-SUONIO, Decentralization of process nets with a centralized control. *Distributed Computing 3*, 2 (1989), 73—87. An earlier version in *Proc. 2nd ACM SIGACT—SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug. 1983, 131—142.

[BK 84] R. J. R. BACK and R. KURI-SUONIO, A case study in constructing distributed algorithms: distributed exchange sort. In *Proc. Winter School on Theoretical Computer Science*, Lammi, Finland, Jan. 1984. Finnish Society of Information Processing Science, 1—33.

[BK 88a] R. J. R. BACK and R. KURKI-SUONIO, Serializability in distributed systems with handshaking. In *Proc. ICALP 88, Automata, Languages and Programming* (Ed. T. Lepistö and A. Salomaa), LNCS 317, Springer-Verlag, 1988, 52—66.

[BK 88b] R. J. R. BACK and R. KURKI-SUONIO, Distributed cooperation with action systems. *ACM Trans. Programming Languages and Systems 10*, 4 (Oct. 1988), 513—554.

[BW 89] R. J. R. BACK and J. VON WRIGHT, Duality in specification languages: a lattice-theoretical approach. Report A 77, Department of Computer Science, Åbo Akademi, 1989. To appear in *Mathematics in Program Construction*, LNCS, Springer-Verlag.

[dB 80] J. DE BAKKER, *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[BG 79] R. BALZER and N. GOLDMAN, Principles of good software specification and their implications for specification languages. In *Specification of Reliable Software*, IEEE Computer Society, 1979, 58—67.

[Ba & 89] F. L. BAUER, B. MÖLLER, M. PARTSCH and P. PEPPER, Formal program construction by transformations — computer-aided, intuition-guided programming. *IEEE Trans. on Software Engineering 15*, 2 (Feb. 1989), 165—180.

[BB 87] T. BOLOGNESI and E. BRINKSMA, Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN System 14*, (1987), 25—59.

[BF 88] L. BOUGÉ and N. FRANCEZ, A compositional approach to superimposition. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, San Diego, California, Jan. 1988, 240—249.

[CM 88a] K. M. CHANDY and J. MISRA, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CM 88b] K. M. CHANDY and J. MISRA, Another view of "fairness". *ACM Software Engineering Notes 13*, 3 (July 1988), 20.

[Di 76] E. W. DIJKSTRA, *A Discipline of Programming*. Prentice-Hall, 1976.

[Di 88] E. W. DISJKSTRA, Position paper on "fairness". *ACM Software Engineering Notes 13*, 2 (April 1988), 18—20.

[DS 80] E. W. DIJKSTRA and C. S. SCHOLTEN, Termination detection for diffusing computations. *Information Processing Letters 11*, 1 (Aug. 1980), 1—4.

[EF 82] T. ELRAD and N. FRANCEZ, Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming 2*, 3 (Dec. 1982), 155—173.

[Fe 87] M. S. FEATHER, Language support for the specification and development of composite systems. *ACM Trans. Programming Languages and Systems 9*, 2 (April 1987), 198—234.

[FD 77] C. FORGY and M. C. DERMOT, OPS, a domain independent production system language. In. *Proc. Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass., Aug. 1977, Morgan Kaufmann, 1977, 933—939.

[Fr 86] N. FRANCEZ, *Fairness*. Springer-Verlag, 1986.

[Ga 89] R. P. GABRIEL (ED.), Draft report on requirements for a common prototyping system. *ACM Sigplan Notices 24*, 3 (March 1989), 93—165.

[Ha 87] D. HAREL, Statecharts: a visual formalism for complex systems. *Science of Computer Programing 8*, 3 (June 1987), 231—274.

[Jä & 89] H.-M. JÄRVINEN, R. KURKI-SUONIO, M. SAKKINEN and K. SYSTÄ, Object-oriented specification of reactive systems. *Proc. 12th International Conference in Software Engineering*, Nice, France, March 1990, IEEE Computer Society Press, 63-71.

[Ka 87] S. KATZ, A superimposition control construct for distributed systems. Microelectronics and Computer Technology Corporation, Report STP—268—87, Aug. 1987.

[Ku 86] R. KURKI-SUONIO, Towards programming with knowledge expressions. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, Jan. 1986, 140—149.

[Ku 89] R. KURKI-SUONIO, Operational specification with joint actions: serializable databases. To appear in *Distributed Computing*.

[KJ 89] R. KURKI-SUONIO and H—M. JÄRVINEN, Action system approach to the specification and design of distributed systems. In *Proc. 5th International Workshop on Software Specification and Design, ACM Software Engineering Notes 14*, 3 (May 1989), 34—40.

[KK 88] R. KURKI-SUONIO and T. KANKAANPÄÄ, On the design of reactive systems. *BIT 28* ,3 (1988), 581—604.

[LS 84] S. S. LAM and A. U. SHANKAR, Protocol verification via projections. *IEEE Trans. on Software Engineering SE—10*, 4 (July 1984), 325—342.

[La 89] L. LAMPORT, A simple approach to specifying concurrent systems. *Comm. ACM 32*, 1 (Jan. 1989), 32—45.

[LF 82] P. E. LONDON and M. S. FEATHER, Implementing specification freedoms. *Science of Computer Programming.2*, 1982, 91—131.

[MP 83] Z. MANNA and A. PNUELI, How to cook a temporal proof system for your pet language. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, Austin, Texas, Jan. 1983, 141—154.

Mo 88a] C. MORGAN, Data refinement by miracles. *Information Processing Letters 26*, (Jan. 1988), 243—246.

Mo 88b] C. MORGAN, The specification statement. *ACM Trans. Programming Languages and Systems 10*, 3 (July 1988), 403—419.

[Mo 87] J. MORRIS, A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming 9*, 3 (Dec. 1987), 287—306.

[Ne 87] G. NELSON, A generalization of Dijkstra's calculus. Res. Rep. 16, Digital Equipment Corporation, Systems Research Center, April 1987.

[Pa 85] D. L. PARNAS, Software aspects of strategic defense systems. *Comm. ACM 28*, 12 (Dec. 1985), 1326—1335.

[Pn 86] A. PNUELI, Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency* (Ed. J. W. de Bakker, W.-P. de Roever and G. Rozenberg), LNCS 224, Springer-Verlag, 1986, 510—584.

[SL 88] F. B. SCHNEIDER and L. LAMPORT, Another position paper on "fairness". *ACM Software Engineering Notes 13*, 3 (July 1988), 18—19.

[SW 87] B. SHRIVER and P. WEGNER (ED.), *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[WZ 88] P. WEGNER and S. B. ZDONIK, Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proc. European Conference on Object-Oriented Programming '88*, Springer-Verlag, 1988, 55—77.