Logical and Operational Methods in the Analysis of Programs and Systems

F. Nielson

together with P. Cousot, M. Dam, P. Degano, P. Jouvelot, A. Mycroft, and B. Thomsen

The **LOMAPS** project studies the use of *Logical and Operational Methods in the Analysis of Programs and Systems* and is sponsored by ESPRIT Basic Research[1]. It has been fostered by the realisation that each of the existing programming paradigms (further discussed below) have their own strengths and weaknesses, and that the reliable construction of computer systems may benefit from a combination of paradigms, and hence the use of multiparadigmatic languages, in order to obtain the modularity, reactivity, and concurrency required. Since each programming paradigm typically has its own set of methods and techniques, this calls for general methods and techniques available over a spectrum of programming paradigms.

The main scientific aims of the project therefore are to conduct basic research into methods and techniques for the analysis of performance and reliability of high-level programs with emphasis on multiparadigmatic languages. This involves hybrid systems that exhibit one or more of the following characteristics: physical distribution, massive parallelism, combinations of computational paradigms (like functional programming and concurrency), higher order communication, and multiple agents with dynamically evolving interconnection topology.

The main technological objectives are the development of advanced analysis and verification techniques for software development. Of particular interest are their application in compilers and programming environments for multiparadigmatic programming languages implemented in a distributed setting. This is important because the market place is already beginning to see the arrival of "end-user" applications that require a multiparadigmatic approach: to handle multiple co-operating agents and to exploit the availability of massively parallel computing systems; examples include multi-media/multi-user systems and real-time process control systems. This development is likely to increase rapidly as access to the internet becomes available to the ordinary consumer, perhaps through dedicated web-browsing hardware to be placed side-by-side with the television and video recorder.

[1] Project number 8130 running from December 1993 to May 1997; for further information consult `http://www.daimi.aau.dk/~bra8130/LOMAPS.html`.

These proceedings take a "snapshot" of the state-of-the-art concerning *Analysis and Verification of Multiple-Agent Languages*, and this introduction relates the "snapshot" to five areas central to the LOMAPS project: Integration of Programming Paradigms [61], Annotated Type and Effect Systems [66], Abstract Interpretation [25], Modalities in Analysis and Verification [30], and Enhanced Operational Semantics [35]. Each section below begins with a presentation of our view of the state-of-the-art within the area, and ends with a brief explanation of how the papers in these proceedings enhance our knowledge of the area.

Integration of Programming Paradigms

Programming notions can be expressed in many different paradigms—imperative, object-oriented, concurrent, functional, logic-programming, constraint, etc. It is widely agreed that each programming paradigm has its own merits and is particularly appropriate for expressing certain classes of computation, thus the choice of paradigm can greatly affect the ease of programming.

Traditionally, when constructing large scale systems, in particular distributed systems, it is often necessary to use multiple programming styles with disparate programming models, and very often it is necessary to resolve conflicts by low level methods reverting to the lowest common denominator. Choosing locally optimal paradigms for each subsystem creates the problem of how to integrate the resulting modules into a coherent whole. The mathematical idea that we could choose a single globally optimal paradigm for expressing the whole system fails to take account of issues like maintenance and the desire for software re-use.

Historically, application developers split their system so that application-specific subsystems depend upon the operating system directly. However, this often causes difficulties with applications not being portable because each operating system has its own distinct application programming interface, and hence different versions of applications may wish to employ different subsystem architectures due to the fact that distinct operating systems provide services that differ in kind and semantics.

Current technology is working to solve these historical problems by introducing a concept called "middleware". Here, an application interfaces with the middleware and normally does not access the underlying operating system directly. One rapidly developing strand in this direction is the tendency of various GUI's (Graphical User Interfaces) such as X-Windows, Windows-95 and Windows-NT to be seen as a low-level interface; instead of providing X over a network one provides (for example) a Java API library. Thus one can see tools which started life as mere web browsers becoming, in effect, a higher level user interface for many systems.

Middleware suppliers strive to present the user with a collection of useful features whose interfaces can be neatly expressed in the various programming languages used for constructing large scale systems. However, little or no attention is paid to integrating these constructs into the programming environment.

To repair this situation several research teams and even a few commercial systems have proposed the notion of coordination languages (e.g. Linda [14], Gamma [6] and LO [4, 5]). The languages are usually not full programming languages, but rather limited to constructs for coordinating communication between component subsystems. Clearly coordination languages solve many of the problems found in the construction of large scale systems by unifying the level of coordination and communication. However, the application programmer is still faced with different programming models for the various modules in the system.

More recently languages that strive to combine several programming paradigms in a single system both for sequential and for distributed computing have been put forward, such as Facile [39, 80], Oz [77], and PICT [69, 70]. A language which supports multiple programming paradigms often enables a more direct expression of the design, since many problems and solutions consist of various components that are more natural and easier when viewed in different ways. Allowing the implementation to exploit a more direct expression of the design helps in avoiding unnecessary encodings that often are great sources of software errors and maintenance complication.

With the emergence on the internet of mobile agents [87, 81] or [41] applets, i.e. chunks of programs that can be sent around a network of computers and execute in different locations, the need for multiple programming paradigms has been further stressed.

It is important however, not to believe that multiparadigmatic programming gives us something for free. There are foundational issues concerned with the ability to express adequately the interface for a subsystem written in one paradigm to another subsystem written in another paradigm. One apposite example is to observe that calls to side-effecting sub-systems cannot directly be expressed in a lazy functional language without risk of losing the reasoning which the functional programmer tacitly assumed. Much work has been done on using "monads" which allow locally side-effecting code to be encapsulated in a manner which preserves functional properties like referential transparency. However, the general question of how the coordination language represents views of interfaces appropriate to each paradigm seems unresolved.

In this volume several papers address, at various levels, the issue of integration of programming paradigms:

Analysis of FACILE programs: a case study (by P. Degano, C. Priami, L. Leth and B. Thomsen) presents a method for analysing mobile agents built in the

Facile language which integrates several programming paradigms. Since mobile agents may carry communication links with them as they move across the network, they create very dynamic interconnection structures that can be extremely complex to analyse. However, since the various paradigms have been integrated carefully with the support of formal semantics, Facile programs may be subjected to formal reasoning. This paper presents a non-interleaving semantics for Facile by looking only at the labels of transitions and then uses the new Facile semantics to debug an agent-based system.

Formalising and prototyping a concurrent object-based language (by L. Fredlund, J. Koistinen and F. Orava) presents a semantics for (a core of) a concurrent object-oriented language by encoding of the language into the polyadic π-calculus. Experimental implementation has been done in Facile. The main concern of the paper is adding constructs for concurrency and distribution to an existing object-oriented language used by Ericsson. The language design and prototype implementation is based on formal semantics to allow for easy experimentation with how the object-oriented paradigm interacts with concurrency and distribution.

Parallel implementation of functional languages (by R. Wilhelm) discusses using a high level lazy functional language to program parallel distributed memory machines. Such machines are difficult to program since they offer a bad ratio of computation speed versus communication speed. Disappointing results on implementing lazy functional languages by parallel graph reductions motivate the design of parallel functional languages (i.e. integration of the concurrent and functional programming paradigms) to facilitate the efficient programming of such machines.

An overview of mobile agent programming (by F. Knabe) presents an overview of mobile agents and an extension of Facile which supports agent programming. The extension takes a step towards making agent programming safer via strong static typing, but many challenges remain. Languages with first-class functions provide a good starting point for agent programming, as they make it easy to express the construction, transmission, receipt, and subsequent execution of agents. However, for developing real agent-based systems, a language implementation must handle architectural heterogeneity between communicating machines and provide sufficient performance for applications based on agents. In addition, agents need to be able to access resources on remote execution sites yet remain in a framework that provides sufficient security. An additional foundational problem is that higher order mechanisms for process (and the paradigm-varying notion of procedure) transmission provide very small-grain interplay between paradigms in contrast to the presumed much larger grained interplay offered by a coordination language.

Annotated Type and Effect Systems

Program analysis offers static techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically during computation. Traditionally this has been used to enable the application of program transformations and to allow compilers to generate more efficient code. This goes beyond merely validating the correctness of the modification; the profitability of performing the modification is also a key issue, and sometimes the modification is profitable only because it opens up for a host of other modifications that more directly improve performance. The presence of higher-order and distributed constructs gives this problem a completely new dimension. Examples include the use of program analysis to assist in documenting the software to be placed in a library of modules, thus helping the programmer in choosing the right module for the task at hand. To fully achieve these goals calls for more research into the theoretical foundations of program analysis as well as their algorithmic (or even decidability) properties.

One of the approaches to program analysis is that of annotated type and effect systems: it requires that a typed programming language be given [76, 45]. The type system is then modified with additional annotations (called effects) that expose further intensional or extensional properties of the semantics of the programs. The literature has seen a great variation in the annotations and effects used. Example effects are: collecting the set of procedures or functions called, collecting the set of storage cells written or read during execution [79], collecting the regions in which evaluation takes place [82]. Other classes of annotations are more ambitious in trying to identify the causality among various operations: that input takes place before output, that communication satisfies protocols as expressed by terms of a process algebra [65]. The interplay between types and effects is brought out when formulating the introduction and elimination rules for the various syntactic constructs (for example function abstraction and function application). Some aspects of a general methodology have emerged but a full understanding of the interplay remains an important research challenge.

The methodology of annotated type and effect systems consists of: *(i)* expressing a program analysis by means of an annotated type or effect system, *(ii)* showing the semantic correctness of the analysis, *(iii)* developing an inference algorithm and proving it syntactically sound and complete. Each of these phases have their own challenges and open problems.

(i) Much research concerns how to incorporate the flow based considerations of context dependent analysis: polyvariance (as opposed to monovariance), k-CFA, "polymorphic" splitting etc.; similarly for the model based considerations of relational (as opposed to independent attribute) analyses and more advanced notions of designing combined property spaces. Although the simple instances of monovariant *0*-CFA analyses in independent attribute form

can be expressed as annotated type and effect systems, it is still unclear how to achieve the more advanced possibilities. Current research suggests that the use of polymorphic recursion (for the annotations, not the types) may be essential and still decidable [82], but the interplay between the use of polymorphism and sub-typing and sub-effecting is still not fully understood [79] and is an important area of further research.

(ii) The techniques needed for establishing semantic soundness are mostly standard. For operational semantics the statement of correctness may be a subject reduction result and the method of proof may benefit from the use of co-induction; when the use of denotational semantics is possible one may benefit from the use of Kripke-logical relations[64].

(iii) Algorithmic techniques often involve the generation of constraint systems in a program independent representation. Sometimes efficient techniques developed for flow based analyses can be used to solve the constraint problems; in other cases the problems take the form of semi-unification problems and then even decidability becomes an issue [82]; furthermore, some applications demand general techniques for algebraic unification. An important area of further research is how to identify those features of the annotated type and effect systems that lead to algorithmic intractability.

In summary, program analysis by means of annotated type and effect systems seems a promising area. The main strength lies in the ability to interact with the user: clarifying what the analysis is about (and when it may fail to be of any help) and in propagating the results back to the user in an understable way (which is not always possible for approaches working on intermediate representations). It will generally be the case that the actual techniques used to obtain efficient implementations require a reformulation of the inference system in more algorithmic terms; however, it is the inference system that focuses on the key properties of the analysis, and hence the implementation must be able to provide feed-back in a form that the user can understand: expressed in terms of the inference system. This is still an area for further research as is the study of the expressiveness of the inference based specifications and the complexity and decidability of the algorithmic realisations.

In these proceedings there are several papers that address goals central to annotated type and effect systems:

The trilogy of papers *Polymorphic Subtyping for Effect Analysis: the Static Semantics* (by H. R. Nielson, F. Nielson, and T. Amtoft), *Polymorphic Subtyping for Effect Analysis: the Dynamic Semantics* (by T. Amtoft, F. Nielson, H. R. Nielson, and J. Ammann), and *Polymorphic Subtyping for Effect Analysis: the Algorithm* (by F. Nielson, H. R. Nielson, and T. Amtoft) all deal with supporting subtyping in polymorphic type and effect systems. This has been the goal of many researchers for quite some time now. Even though some success had been previously reported, in particular via the notion of "subeffecting", which however limited the static order relationship to the effect domain, the full generality of subtyping was still a challenge. This is what the trilogy of papers addresses and solves. While the first paper presents a general overview of the integration process, the two subsequent ones address more technical, though equally important, questions.

The *Static Semantics* paper describes a new static semantics that approximates some aspects of the behavior of Concurrent ML programs. In particular, for any expression, this semantics specifies a polymorphic ML-like type and an effect that gives an upper-approximation on what kind of channels are allocated during the expression evaluation. This annotated type system supports polymorphism, effects and subtyping and is the first to integrate these three aspects in a rigorous and compatible way. Applications of the presented approach to other effect systems, in particular memory effects, should pose no major problems. Effect systems could thus be used to cleanly specify the type generalization process in ML-like languages without restricting it in odd ways imposed by the effect system technology.

The *Dynamic Semantics* paper shows that the static semantics, which approximates some aspects of the behavior of Concurrent ML programs (and which is described in "the Static Semantics" paper), is sound with respect to an operational concurrent semantics. The main theorem is akin to a subject reduction one and expresses that evaluation preserves, in a sense made precise in the paper, types and behaviors, thus ensuring the semantic soundness of the static type and effect system.

The *Algorithm* paper is the last in this trilogy, and describes a Milner's W-like reconstruction algorithm for the new static semantics introduced in the two companion papers. This algorithm is inspired by previous techniques published in the literature but introduces a couple of new ideas, in particular the notion of closures over a set of constraints. It is proved sound (but not complete) with respect to the static semantics described above.

Polyvariance, Polymorphism and Flow Analysis (by K.-F. Faxén) considers determining an approximation of the set of possible values (mainly functions) expressions can have. This kind of analysis is of paramount importance in optimizing implementations of functional languages, since environment management for closures requires rather heavy machinery. Using ideas taken from the theory of type systems, control-flow analysis, set-based analysis, constraint systems, and polyvariant analysis, the paper suggests a family of heuristics for performing such flow analysis for untyped languages. Although the system is not yet implemented, examples are provided to support the claim that such analyses could be used for performing useful optimizations.

Type inference for a multiset rewriting language (by P. Fradet and D. Le Métayer) considers the multiset rewriting language Gamma extended with a notion of addresses and contents of addresses. This allows to code structured multisets corresponding to algebraic data structures and enhances the ease with which programming can be performed in Gamma. Context-free graph grammars are then used to define the shape that the structured multisets are supposed to have. The paper then develops a sound (but not complete) type checking algorithm for ensuring that each rewrite (allowed by the Gamma program) does in fact maintain the invariant expressed by the graph grammar.

Abstract Interpretation

Abstract interpretation aims at gathering information about programs by approximation of their runtime behavior. The point of view is that to automatically answer questions about programs one cannot only consider a single semantics but better a hierarchy of semantics each one adapted to a category of program properties. Abstract interpretation is a formalization of the relationships between these semantics and their combinations. In particular the abstraction, and its inverse, the refinement of a semantics are essential concepts used to partially order semantics in this hierarchy [24].

These ideas of discrete approximation are involved in formal semantics [18]. For example an operational trace semantics can be approximated by a denotational semantics by forgetting about intermediate states. A further approximation into a big step operational semantics essentially consists in forgetting about nontermination and in reformulating fixpoint definitions in equivalent rule-based form [23].

These ideas of discrete approximation are involved in proof methods [17]. For example the Floyd/Hoare/Owicki/Lamport proof method essentially consists in approximating sets of execution traces by sets of states, such invariants being adequate for safety but essentially incomplete for liveness properties.

These ideas of discrete approximation are central to program analysis methods [19]. A considerable number of program analysis methods have been designed (dataflow analysis, strictness analysis, termination analysis, closure analysis, projection analysis, binding-time analysis, set-based analysis, type inference, effect systems, lazy types, etc., to cite a few finitary methods). The purpose of abstract interpretation is to unify these superficially different methods in a single mathematical framework by understanding all of them as approximations of semantics using specific abstract domains to encode approximate program properties. This point of view often leads to cross-fertilization. For example understanding set-based analysis as an abstract interpretation yields a powerful generalization based upon context-sensitive tree grammars [22]. Formalization by abstract interpretation often leads to simplifications. For example, several independent analyses can be combined within the same abstract framework such as comportment analyses [21, 60, 84], the combination being more precise than the individual analyses. An important recent contribution has been the understanding of type inference as abstract interpretation [26, 56, 58, 57]. Another recurrent theme in abstract interpretation is the design of general approximation methods independently of a particular application. Essentially two classical methods are used, static ones, often based on Galois connections, where the approximation is fixed while defining the abstract properties for a programming language and dynamic ones, often based on widening operators, where the ultimate approximation is performed during the analysis process [20]. The Galois connection based abstract interpretation framework has been extended to higher-order functional languages [13, 67], by a clear separation of the computational and approximation ordering, both at the concrete and abstract level [21, 62, 59, 46]. The dynamic approximation is essentially more powerful than the static one, since it allows for infinite abstract domains, expressing non-uniform properties. The design of such very expressive infinite abstract domains with their associated widening operators is a continuous research topic in abstract interpretation, because they leads to extremely powerful program analyses [55, 51, 85], hardly achievable by the above mentioned finitary methods. Another application is the analysis of parallel, centralized [27] or distributed programs [42] e.g. for model-checking [27] or to automatically generate schedulers [42] by abstract interpretation of a truly-concurrent semantics using Higher-Dimensional Automata, in which the level of concurrency is specified by the dimension of the transitions that can be fired. Applications range from verification of protocols for distributed systems on a given architecture to the dual problem of parallelizing programs.

Proving Properties of Logic Programs by Abstract Diagnosis (by M. Comini, G. Levi, M.-C. Meo, and G. Vitiello) explore the verification of logic programs by comparison of an abstract semantics with an abstract specification, thus generalizing declarative diagnosis where the declarative semantics of a program is compared to a specification of its intended behavior. In particular, when they differ, the program components which are sources of errors must be determined. The paper introduces several abstract semantics and identifies corresponding conditions for the abstract diagnosis to be complete. In general however, the specification is infinite and some approximation is necessary. In this case, partial results are obtained. For example uncovered elements always correspond to a program bug while incorrect clauses provide a hint to locate possible bugs. Several useful abstractions are proposed.

Implementing a Static Analyzer of Concurrent Programs: Problems and Perspectives (by R. Cridlig) discusses the implementation of a prototype analyser of asynchronous shared-memory Concurrent Pascal programs. The truly parallel semantics using Higher Dimensional Automata is approximated into finite abstract automata by foldings by means of data-dependent widenings. The experiments show an important state space reduction of over 90%; only in one case is model-checking not feasible due to loss of information (in which case the concrete transition system explodes anyway).

Abstract Interpretation of Small-Step Semantics (by D.A. Schmidt) considers the analysis of CCS (and extensions such as channel creation): model-checking on a conservative approximation of the program semantics. Besides the use of small-step environment semantics, a characteristic of the analysis is the regular-expression-like abstract interpretation of the syntactic encoding of the program configurations (including the control and communication components), which is of general use for analysing programs with SOS semantics [71] (Structural Operational Semantics).

Abstract Interpretation of the π-Calculus (by A. Venet) is concerned with the analysis of the distribution of processes and the channel communication topology for systems of mobile processes. The analysis is based on a refinement of the classical operational semantics given by a structural congruence and a process reduction relation. This refined semantics allows for a direct way to compute the internal communication topology associated to a configuration, whereas, in the standard semantics, it is indirectly encoded within a process algebra. The abstraction uses an infinite abstract cofibrated domain with widening, originally introduced by the author for alias analysis. This abstract domain allows for tractable although undecidable analyses, with precise non-uniform distinction between instances of recursively spawned and untyped processes.

Modalities in Analysis and Verification

Typically, modal and temporal operators are used in program logics and types to express properties relating to dynamic behaviour, i.e. state change capabilities. Modal operators are used for local properties, for instance the fact that a transition leading to some desired state is enabled at the present state. Temporal operators describe properties of complete computations, for instance the existence of future states with some desired properties (like being terminal). Models are transition systems determined, for instance, by an SOS semantics. States are transition system states with a control component (typically the abstract program currently under evaluation) and maybe a store determining bindings of values to identifiers. Transitions may carry information relevant for synchronisation and parameter passing as in CCS, CSP or the π-calculus, and they may carry finer information as they do in proved transition systems [33].

A direct strategy for verifying programs against their modal and temporal specifications is thus to explore the transition graph or computation tree that programs give rise to by means of their operational semantics when started in a given initial state. If this graph is finite then most temporal queries can be answered by traversing and marking the graph using model checking. Alternatively, in an axiomatic setting, for each reachable control point of the program under consideration a unique constant is introduced to axiomatise the one-step computation relation. To a limited extent these approaches can be extended to infinite state programs. For model checking this is possible when the property under consideration depends only on a finite portion of an otherwise infinite state graph (cf. [3]), or where infinite domains can be given finite representations as in the π-calculus [29, 2]. Axiomatic approaches can permit infinite data domains in more general terms, as long as the number of control points remain bounded (c.f. [75]).

Even though these approaches work extremely well in many circumstances, a basic problem is that state spaces only rarely stay small enough for exhaustive traversal to be computationally feasible, or at all possible. One difficulty is the well-known state explosion problem that n parallel processes each with 2 states have 2^n global states. This problem has received much attention over the past years, in the LOMAPS project and elsewhere (c.f [36, 83, 88]). Related problems concern value-passing processes. Even for finite domains the state spaces can quickly become too large to be manageable. Options for addressing this issue is to use symbolic methods [29] or techniques based on abstract interpretation [19].

Not only value passing and combinatorial state explosion are sources of problems, however. A great many control constructs give rise to unbounded growth of state spaces. In a few cases (cf. [12, 86]) this growth can be checked by algorithmic means. Even for very simple parallel programs, however, decidability is lost [38]. This problem needs attention in particular for multiparadigmatic languages that combine concurrency and distribution with a "first-class" treatment of abstraction and communication. At the heart of the problem is the ability to dynamically create and communicate new processes. Furthermore, in many modern modelling and programming languages the distinction between data and control is becoming blurred: π-calculus is one example, and higher-order process communication is another.

Many approaches are being explored to deal algorithmically with large and infinite state spaces, including techniques based on BDDs (Binary Decision Diagrams) and their successors [11], techniques that exploit process symmetries (cf. [37]), and partial order techniques (cf. [88]). Further state space reduction may be obtained by resorting to approximate techniques such as abstract interpretation. This is not only a formal activity: Preceding a formal analysis is often an "abstraction" phase in which the problem at hand is modelled and simplified, for instance to make automatic analysis feasible (by limiting attention to, say, a finite value domain, or for protocol analysis to the case of a single sender and a single receiver). Once a formal model is obtained, further abstractions may be possible, for instance by collapsing "similar" states as in techniques based on abstract interpretation in the papers by Cridlig, Schmidt, and Venet (in the present volume), or, say, state space hashing as in Holzmann's SPIN system [43].

While fully automated, state space exploration-based approaches have the very important virtue that they require no user intervention (up to the modelling and abstraction phase referred to above), they also face some serious problems:

- Limited scope: Fundamentally the scope of these techniques is limited. Language constructs such as dynamic process creation and higher-order communication can not in general be accomodated if exact analyses are called for.
- Code verification: As a result of the previous point much emphasis needs to be put on an initial, informal modelling and abstraction phase, to force problems into a tractable framework. Thus code- (as opposed to model-) verification will often be difficult to accomodate.
- Scalability/feasibility: Even with very smart state space compression techniques the state space explosion problem quickly makes its presence felt as parallel components are added, or as the size of value domains is increased.
- Modularity/reusability: Often actual systems are constructed from reusable building blocks, or modules, and it is really properties of modules rather than actual system configurations that are of interest. However, even though actual configurations may well be finite-state, modules are not easily so represented: They are really *open* systems, designed to operate in environments that are not yet fully instantiated.
- Approximate analysis: The use of approximate techniques comes with a cost, namely that it may be difficult to interpret analysis results.

An alternative, then, is to accept undecidability and resort instead to some sort of theorem proving. Many frameworks have been proposed for this: Tableaux [53], Hoare-style proof systems based on the rely-guarantee paradigm (cf. [15]), embeddings into a higher-order type theory as in the PVS system [68]. The scope of deductive techniques is certainly in principle greater than that of algorithmic ones. However it is also clear that, state explosion will remain a problem, and certainly even in an interactive framework one will still need automated procedures to deal with the large amounts of trivia that verification of "real" systems gives rise to. Indeed the combination of deductive and automatic techniques is currently receiving considerable attention in the literature (c.f. [68, 75]). To deal with features like openness, dynamic process creation, and higher-order communication a compositional approach to verification seems indispensable. In [28] an approach was introduced with the scope, in principle, of addressing systems with such features. In this work general proof principles were identified for a proof-based compositional handling of temporal properties of dynamic process networks. The approach is based on a kind of "internalised abstraction" originating with Stirling [78]: Instead of proving directly an assertion such as $p \parallel q : \phi$, abstractions of p and q are provided as properties ψ and γ, and proof obligations are created to show the abstractions correct (to show $p : \psi$, $q : \gamma$, and that $x \parallel y : \phi$ whenever $x : \psi$ and $y : \gamma$). The difficulty is dealing in an adequate way with temporal properties. The work has been extended to the π-calculus (c.f. [2, 31]) and, at least partially, to higher-order processes [1]. However, much work remains to be done before we can truly claim that verification of open, dynamic, and higher-order process networks is feasible, theoretically as well as practically. It is certainly clear that both automated and interactive techniques must be brought to bear if this goal is to be realised.

Enhanced Operational Semantics

Since the very beginning of computer science, the behaviour of machines has been given through an operational approach which describes the transitions between states that a machine performs while computing. A graphical representation of behaviours as oriented graphs, usually called *transition systems*, is quite easy: nodes represent the set of states that the machine can pass through, and arcs, possibly labelled, denote the transitions between states.

The term operational semantics appeared in the literature during the sixties due to [52, 48]. A program is seen as a sequence of atomic instructions that operate on the states of the machine. States consists of the program itself and some auxiliary data which can represent the store or the data structure on which the program works. Then, a function from states to states indicates the moves from one given configuration to another. These transitions may be labelled by additional information on the activity performed. Finally, a run of a program (or computation) is represented through a sequence of states where each state is connected to the next one through the transition function. The last state of the sequence, if any, is the final configuration of the machine after the execution of the program. But we need also in some situations to be able to describe infinite computations. Operating systems, and some reactive systems should precisely never terminate, and we should be able to describe their infinite computations at least under some fairness conditions. Interesting infinite computations also appear in (interleaving) concurrency when a non-bounded number of processes can be put in parallel. Lazy functional languages also exhibit natural infinite computations (to deal with lazy infinite lists for instance).

A renewed interest in operational semantics is due to Plotkin, and to his formal method, called SOS for *Structural Operational Semantics* [71, 63]. The key idea is to deduce transitions by inducing on the syntactic structure of the machine itself. States are essentially programs expressed according to the abstract syntax of the considered language defined through a *BNF*-like grammar. In this way, one exploits the duality between languages and abstract machines. Transitions are then defined by a set of rules that induce on the abstract syntax. The inference rules have the following form

$$Premises \Longrightarrow Conclusion.$$

Its meaning is that when the premises are satisfied, the conclusion is satisfied as well. Operationally, the above rule says that when the computational steps corresponding to the premises occurred, the one corresponding to the conclusion is enabled.

Operational semantics is mathematically simple and is close to intuition, thus it gives guidelines to implementations. In fact, it describes the essential features that any computing device has. So also untrained people can grasp the meaning of a definition on the basis of their experience with their own machine. The inductive definition of transitions naturally suggests to use induction for proving properties of programs. This approach is better suited than others to cope with programming languages that include heterogeneous features. In fact, it has been successfully used to describe imperative, functional, logic, object-oriented, concurrent and distributed languages.

Within the LOMAPS project we study enhancements of structural operational semantics along several lines. The first relies on the observation that the labels of transitions can give an expressive and detailed description of the behaviour of complex systems; so we proposed to decorate transitions with structured, rich information. The second main line of research takes advantage of the structure of states; then, the transitions themselves maintain this structure, or rather, only the part of it relevant to the aspects of a system to be described. A third topic concerns the specification of infinite computations; for that purpose, $G_\infty SOS$ combines the inductive definition of finite behaviors and the simultaneous co-inductive definition of infinite behaviors within a unique bi-inductive interpretation of inference rules [18], which is preserved by abstract interpretation [23].

We start illustrating our first line, mainly discussing its applications to concurrency. Essentially, we propose a general, structural operational approach to semantics that can be easily instantiated to cover the various aspects relevant to build concurrent and distributed systems. More generally, we think that semantics offers firm grounds to the many distinct activities for producing software. The logical nature of structural operational semantics, and specifically the proofs of transitions, gather (almost) all the information needed in the various phases of system development, e.g. design, implementation, quality control, management, etc. This information can be roughly grouped in two. The first concerns the *behavioural* or *qualitative* aspects of systems, i.e. in *what* they do, regardless of how. The other kind of information has to do with the *quantitative* aspects of systems, i.e. *how* efficiently they perform.

Different views of the same system originated many different semantics that must be related to one another via a strict correspondence among them all.

For example, concurrent and distributed processes have been given descriptions that take care of aspects like causality and locality (so-called *true concurrency*), as well as priorities, time, probabilities. These descriptions, together with the classic interleaving ones, help improving the quality and robustness of code and efficient runtime management of systems. We think that there is no need of many semantic models defined ad hoc: a single *parametric* model can capture all aspects of interest, both qualitative and quantitative. The main semantics presented in the literature can be retrieved, by simply projecting the parametric model on the properties under investigation. The connections among different semantics are now easy to establish: comparison of projections suffice.

We implement the motto TRANSITIONS AS PROOFS through *proved transition systems* [32, 10]. Their transitions are labelled by encodings of their proofs. Then, suitable relabellings yield the wanted models, that are related to each other by comparing their relabelling functions [33, 34, 73, 9]. Also, proved transition systems can originate a hierarchy of semantic descriptions of the same process that are closer and closer to actual implementations. The parallel structure of processes, i.e. the network topology, made explicit by in the proofs of transitions, is exploited in [8] for transforming global environments of mobile processes into local ones.

Performance evaluation and other quantitative analysis should start already at the design level. Besides robustness and reliability of the design, these early measures may save efforts. Indeed, if the design meets all behavioural requirements but leads to inefficient implementations, the system must be re-designed. This calls for the integration of qualitative and quantitative analysis of systems in a single methodology. Proved transition systems are detailed enough to express these quantitative aspects; see [72, 74] for descriptions that include probabilistic aspects of concurrent systems, and these proceedings for a model that takes time into account.

In most timed operational models, time is assumed to be discrete, i.e. to be a multiple of some base time. This discrete approach cannot certainly account for continuous phenomena arising in hybrid systems, and runs into a complexity problem when put at work in a program analysis context when the base time is chosen as the greatest common divisor of the execution times of all actions considered in the semantics. When taking the refinement process up to its limit, dense time abstracts all possible base times. We have introduced dense time in the context of Higher-Dimensional Automata, a truly-concurrent model of computations [40] in which transitions are made up of several concurrent activities. Time is measured as the length of traces using some metric (defined locally by the norm of tangent vectors). The timed semantics can be defined compositionally (using constructors much alike those of timed process algebras). A SOS-like format has also been proposed and proved correct with respect to the underlying timed Higher-Dimensional Automata.

Our "economic" long-run goal is the realization of (semi-) automatic verification tools to be integrated in a programming environment for concurrency [16]. Again parametricity saves efforts. In fact, the implementation of verification tools is complex and expensive. If the tool is highly specialized, its cost may not be justified. Instead, parametric tools support many different semantics at the cost of a single one. Its specializations simply require to implement relabelling functions [44]. Indeed, the kernel of a parametric tool coincides with the construction of the proved transition system. It has the same shape and the same space and time complexity of the interleaving case. Then, model checking and (bisimulation-based) equivalence verifications are completely standard. We implemented a prototype of an equivalence checker based on proved transition systems [7]. It has also been used to support the debugging of *Facile* code [9] (and below).

Semantics expressed by means of transition systems and sets of rules to construct them have been proposed also for Statecharts, a formalism for the specification of systems consisting of concurrent components which react to signals from an environment and interact with each other by broadcast. Due to the complexity of specifications compositionality of semantic descriptions, concepts of equivalence of behaviours, composability of proof methods are mandatory ([49, 50, 47]).

In these proceedings there are several papers that address goals central to annotated type and effect systems:

A non-standard semantics for generating reduced transition systems (by N. De Francesco, A. Santone and G. Vaglini) proposes a new non-standard operational semantics for CCS that gives a transition system with much less transitions and states than the original one. The authors apply abstract interpretation to obtain their non-standard operational semantics, and prove that the new transition system of a process is deadlock free if and only if the standard one is. The key point is that the reachability relation of the standard transition system is preserved by the abstract interpretation they define.

Mobile processes with local clocks (by P. Degano, J.-V. Loddo, and C. Priami) defines a structural timed operational semantics for mobile processes. A distinguishing feature is that the time that actions consume depends on the basic operations needed for firing them. In this way, both the run-time support and the architecture of the system affect the estimate of time. Their proposal is truly concurrent, because any sequential component of a system has its own clock. A preorder is then defined that considers a process more efficient than another if it performs better from a given point on.

Testing semantics of asynchronous distributed programs (by R. De Nicola and R. Pugliese) carries over an existing language some important notions and concepts developped mainly for theoretical process calculi. It defines a structural operational semantics to IPAL, an imperative subset of the coordination language Linda. Essentially, this subset is a CSP-like process calculus. The main results of the paper are that assignment and Linda communication, based on tuples, are given a transitional semantics. Also, testing equivalence is extended to cover IPAL, a proof system is given, and a full abstraction theorem is proved.

The last two two papers address semantic and expressivity issues of Statecharts.

A process language for statecharts (by F. Levi) proposes a compositional labelled transition system for Statecharts. This new semantics is obtained via a translation of Statecharts into a process algebra, which can be seen as the counterpart of Statecharts. Notably, the process algebra used has an operator of process refinement that represents the hierarchical structure Statecharts have. The semantics proposed is shown correct with respect to Pnueli and Shalev's.

Priorities in Statecharts (by A. Maggiolo-Schettini and M. Merro) deals with the capability of the formalism to express notions of priority between transitions that are enabled and are mutually exclusive. A version of Statecharts without priority is introduced, and then is extended with various syntactic and semantic notions of priority. These are examined and classified according to their expressive power.

References

1. R. Amadio and M. Dam. Reasoning about higher-order processes. In *Proc. CAAP'94,* Lecture Notes in Computer Science, 915:202–217, 1995.
2. R. Amadio and M. Dam. A modal theory of types for the π-calculus. In *Proc. FTRTFT'96,* Lecture Notes in Computer Science, 1135:347–365, 1996.
3. H. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ-calculus. In Proc. LICS'94, 1994.
4. J.-M Andreoli and R. Pareschi: "Linear objects: Logical processes with built-in inheritance". In D.H.D. Warren and P. Szeredi, editors, 7th Int. Conf. Logic Programming. MIT Press, 1990.
5. J.-M Andreoli, R. Pareschi, L. Leth and B. Thomsen: "True Concurrency Semantics for a Linear Logic Programming Language with Broadcast Communication". In proc. Conf. on Theory and Practice of Software Development (TAPSOFT'93), vol 668 of LNCS, pp. 182-198. Springer Verlag, 1993.
6. J.-P. Banatre and D. Le Metayer. "Programming by multiset transformations. CACM, 36(1):98, 1993
7. A. Bianchi, S. Coluccini, P. Degano, and C. Priami. An efficient verifier of truly concurrent properties. In V. Malyshkin, editor, *Proceedings of PaCT'95, LNCS 964*, pages 36–50. Springer-Verlag, 1995.
8. C. Bodei, P. Degano, and C. Priami. Mobile processes with a distributed environment. In *Proceedings of ICALP'96, LNCS 1099*, pages 490–501. Springer-Verlag, 1996.
9. Roberta Borgia, Pierpaolo Degano, Corrado Priami, Lone Leth, and Bent Thomsen. Understanding mobile agents via a non-interleaving semantics for Facile. In

R. Cousot and D.A. Schmidt, editors, *Proceedings of SAS'96*, LNCS 1145, pages 98–112. Springer-Verlag, 1996. Extended version in European Computer-Industry Research Center Tech. Rep. ECRC-96-4, 1996.

10. G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, XI(4):433–452, 1988.
11. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Proc. LICS'90*, 1990.
12. O. Burkart and B. Steffen. Model checking for context-free processes. In *Proc. CONCUR'92,* Lecture Notes in Computer Science, 630:123–137, 1992.
13. G. L. Burn, C. Hankin, and S. Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7:249–278, 1986.
14. N. Carriero, D. Gelernter, T. Mattson and A. Sherman. "The Linda alternative to message passing systems". Parallel Computing 20(4):633-655, April 1994.
15. A. Cau and P. Collette. Parallel composition of assumption-commitment specifications. *Acta Informatica*, 33:153–176, 1996.
16. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transaction on Programming Languages and Systems*, pages 36–72, 1993.
17. P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier, 1990.
18. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *19th POPL*, pages 83–94. ACM Press, 1992.
19. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, 1992.
20. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. Int. Work. PLILP '92.* LNCS 631, pages 269–295. Springer-Verlag, 1992.
21. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. 1994 ICCL*, pages 95–112. IEEE Comp. Soc. Press, 1994.
22. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7th FPCA*, pages 170–181. ACM Press, 1995.
23. P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper. In P. Wolper, editor, *Proc. 7th Int. Conf. CAV '95.* LNCS 939, pages 293–308. Springer-Verlag, 1995.
24. P. Cousot, R. Cousot, and A. Mycroft. Report on a Dagsthul seminar on abstract interpretation, 1995.
25. P. Cousot: Abstract Interpretation. *Computing Surveys* **28** *2*, pages 324–328, ACM Press, 1996.
26. P. Cousot. Types as abstract interpretation. In *24th POPL.* ACM Press, 1997.
27. R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proc. PEPM '95.* ACM Press, 1995.
28. M. Dam. Compositional proof systems for model checking infinite state processes. In *Proc. CONCUR'95,* Lecture Notes in Computer Science, 962:12–26, 1995.

29. M. Dam. Model checking mobile processes. *Information and Computation*, 129:35–51, 1996.
30. M. Dam: Modalities in Analysis and Verification. *ACM Computing Surveys* **28** *2*, pages 346–348, ACM Press, 1996.
31. M. Dam. Compositional verification of mobile process networks. In preparation, 1997.
32. P. Degano, R. De Nicola, and U. Montanari. Partial ordering derivations for CCS. In *Proceedings of FCT, LNCS 199*, pages 520–533. Springer-Verlag, 1985.
33. P. Degano and C. Priami. Proved trees. In *Proceedings of ICALP'92, LNCS 623*, pages 629–640. Springer-Verlag, 1992.
34. P. Degano and C. Priami. Causality for mobile processes. In *Proceedings of ICALP'95, LNCS 944*, pages 660–671. Springer-Verlag, 1995.
35. P. Degano and C. Priami: Enhanced Operational Semantics. *Computing Surveys* **28** *2*, pages 352–354, ACM Press, 1996.
36. P. Degano and C. Priami: A Compact Representation of Finite State Processes. Report available via `http://www.daimi.aau.dk/~bra8130/LOMAPS_papers.html` by selection of LOMAPS-DIPISA-2.
37. E. A. Emerson and A.P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.
38. J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 1996. (to appear).
39. A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18:121–160, 1989.
40. E. Goubault. Durations for truly-concurrent actions. In *Proceedings of ESOP'96, LNCS, 1058*, pages 173–187. Springer-Verlag, 1996.
41. J. Gosling and H. McGilton. The Java language environment. White paper, May 1995. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA.
42. E. Goubault. Schedulers as abstract interpretations of higher-dimensional automata. In *Proc. PEPM '95*, La Jolla, Calif., 21–23 jun 1995, pages 134–145. ACM Press, jun 1995.
43. G. J. Holzmann. An analysis of bitstate hashing. In *Proc. PSTV'95,* Chapman and Hall, pages 301–314, 1995.
44. P. Inverardi, C. Priami, and D. Yankelevich. Automatizing parametric reasoning on distributed concurrent systems. *Formal Aspects of Computing*, 6(6):676–695, 1994.
45. M. P. Jones. A theory of qualified types. In *Proc. ESOP '92*, pages 287–306, Springer Lecture Notes in Computer Science **582**, 1992.
46. N. D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science volume 4*. Oxford University Press, 1995.
47. F. Levi. Verification of Temporal and Real-Time Properties of Statecharts. PhD Thesis in Computer Science, University of Pisa, to be discussed in January 1997.
48. P. Lucas. Formal definition of programming languages and systems. In Springer Verlag, editor, *Proceedings of IFIP'71*, 1971.
49. A. Maggiolo-Schettini and A. Peron Retiming Techniques for Statecharts In Proc. FTRTFT '96, LNCS 1135, pages 55–71. Springer-Verlag, 1996.
50. A. Maggiolo-Schettini, A. Peron and S. Tini Equivalences of Statecharts In Proc. CONCUR '96, LNCS 1119, pages 687–702. Springer-Verlag, 1996.

51. L. Mauborgne. Abstract interpretation using TDGs. In B. Le Charlier, editor, *Proc. SAS '94*, Namur, 20–22 sep 1994, LNCS 864, pages 363–379. Springer-Verlag, 1994.
52. J. McCarthy. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Information Processing 1962*, pages 21–28, 1963.
53. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
54. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (I and II). *Information and Computation*, 100(1):1–77, 1992.
55. B. Monsuez. Polymorphic types and widening operators. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Proc. 3rd Int. Work. WSA '93 on Static Analysis.* LNCS 724, pages 267–281. Springer-Verlag, 1993.
56. B. Monsuez. Polymorphic typing for call-by-name semantics. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA.* LNCS 735, pages 156–169. Springer-Verlag, 1993.
57. B. Monsuez. System F and abstract interpretation. In A. Mycroft, editor, *Proc. SAS '95.* LNCS 983, pages 279–295. Springer-Verlag, 1995.
58. B. Monsuez. Using abstract interpretation to define a strictness type inference system. In *Proc. PEPM '95*, pages 122–133. ACM Press, 1995.
59. A. Mycroft and F. Nielson. Strong Abstract Interpretation using Power Domains. In *Proc. ICALP '83*, volume 154, pages 536–547. SLNCS, 1983.
60. A. Mycroft and K.L. Solberg. Uniform PERs and comportment analysis. PLILP'95, 1995.
61. A. Mycroft: On Integration of Programming Paradigms. *ACM Computing Surveys* **28** *2*, pages 309–311, ACM Press, 1996.
62. F. Nielson. Two-Level Semantics and Abstract Interpretation. *Theoretical Computer Science — Fundamental Studies*, 69:117–242, 1989.
63. F. Nielson and H.R. Nielson. *Semantics with applications: a formal introduction.* Wiley, 1992.
64. F Nielson and H.R. Nielson. Layered Predicates. In *Proc. REX'92 workshop on "Semantics—foundations and applications"*, pages 425–456, Springer Lecture Notes in Computer Science **666**, 1993.
65. H. R. Nielson and F. Nielson. Higher-Order Concurrent Programs with Finite Communication Topology. In *Proc. POPL '94*, pages 84–97, ACM Press, 1994.
66. F. Nielson: Annotated Type and Effect Systems. *ACM Computing Surveys* **28** *2*, pages 344–345, ACM Press, 1996.
67. F Nielson. Semantics-Directed Program Analysis: A Tool-Maker's Perspective. In *Proc. SAS'96*, pages 2–21, Springer Lecture Notes in Computer Science **1145**, 1996.
68. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. CAV'96,* Lecture Notes in Computer Science, 1102:411–414, 1996.
69. B. Pierce and D. Turner. PICT Language Definition. University of Indiana, December 1995.
70. B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In Theory and Practice of Parallel Programming, volume 907 of Lecture Notes in Computer Science. Springer-Verlag, April 1995.
71. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.
72. C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(6):578–589, 1995.

73. C. Priami. *Enhanced Operational Semantics for Concurrency.* PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1996. Available as Tech. Rep. TD-08/96.
74. C. Priami. Integrating behavioural and performance analysis with topology information. In *Proceedings of 29th Hawaian International Conference on System Sciences*, volume 1, pages 508–516, Maui, Hawaii, 1996. IEEE.
75. H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In *Proc. CAV'96,* Lecture Notes in Computer Science, 1102:208–219, 1996.
76. G. S. Smith. Principal Type Schemes for Functional Programs with Overloading and Subtyping. *Science of Computer Programming* **23**, pages 197-226, 1994.
77. G. Smolka. The definition of kernal Oz, in Constraints: Basic and Trends, Lecture Notes in Computer Science 910. Springer Verlag, 1995.
78. C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–347, 1987.
79. J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation* **111**, pages 245–296, 1994.
80. B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In Proceedings of CONCUR'96 - Seventh Intl. Conf. on Concurrency Theory, volume 1119 of Lecture Notes in Computer Science, pages 278-298. Springer-Verlag, 1996.
81. B. Thomsen, F. Knabe, L. Leth and P.-Y. Chevalier. Mobile Agents Set to Work, In Communications International, July, 1995.
82. M. Tofte and J.-P. Talpin: Implementation of the Typed Call-by-Value λ-Calculus using a Stack of Regions. In *Proc. POPL '94*, pages 188–210, ACM Press, 1994.
83. A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1992.
84. F. Védrine. Binding-time analysis and strictness analysis by abstract interpretation. In A. Mycroft, editor, *Proc. SAS '95.* LNCS 983, pages 400–417. Springer-Verlag, 1995.
85. A. Venet. Abstract cofibred domains: Application to the alias analysis of untyped programs. In R. Cousot and D.A. Schmidt, editors, *Proc. SAS '96.* LNCS 1145, pages 368–382. Springer-Verlag, 1996.
86. I. Walukiewicz. Pushdown processes: Games and model checking. In *Proc. CAV'96,* Lecture Notes in Computer Science, 1102:62–74, 1996.
87. J. E. White. "Telescript technology: The foundation for the electronic marketplace". General Magic white paper, 2465 Latham Street, Mountain View, CA 94040, 1994.
88. P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proc. CONCUR'93,* Lecture Notes in Computer Science, **715**:233–246, 1993.