Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

335

F. H. Vogt (Ed.)

CONCURRENCY 88

International Conference on Concurrency
Hamburg, FRG, October 1988
Proceedings

Springer-Verlag

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

F. H. Vogt (Ed.)

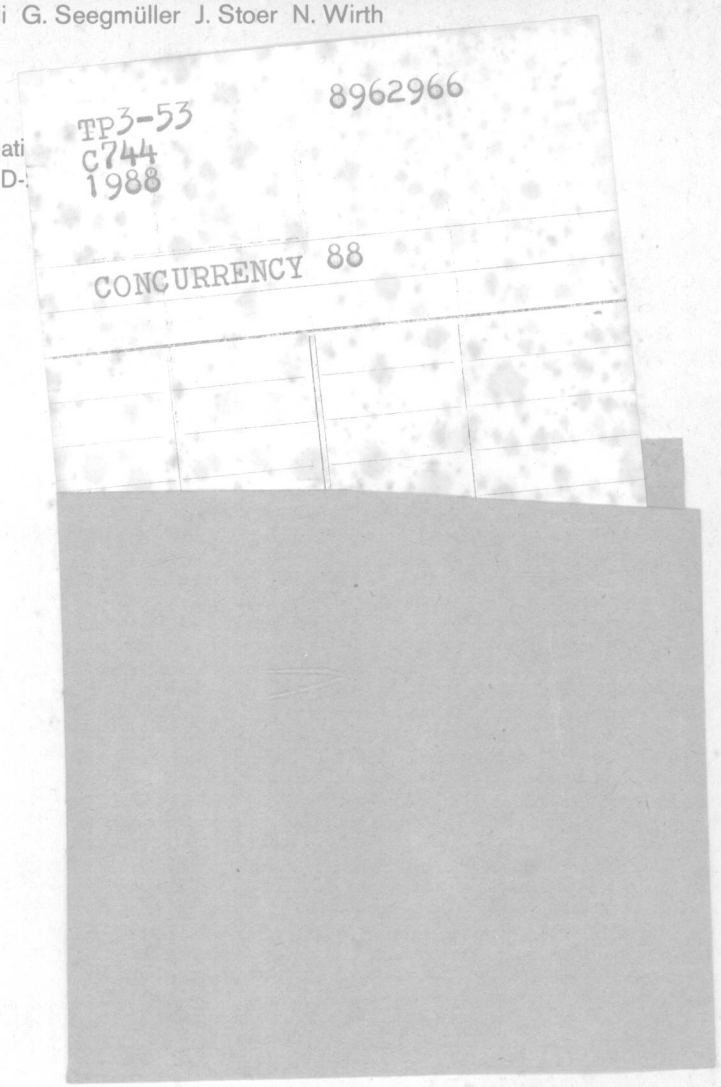
CONCURRENCY 88

International Conference on Concurrency
Hamburg, FRG, October 18–19, 1988
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

Editor

Friedrich H. Vogt
Fachbereich Informati
Bodenstedtstr. 16, D-



CR Subject Classification (1987): D.1.3, D.2.1, D.2.4, F.3.1–2, H.2.4, I.2.2

ISBN 3-540-50403-6 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-50403-6 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210

Foreword

CONCURRENCY 88, the international conference on formal methods for distributed systems, was held October 18-19, 1988, in Hamburg, Federal Republic of Germany.

This conference was presented in connection with the annual conference of the German Society for Computer Science (GI), dedicated this year to complex and distributed systems.

CONCURRENCY 88 was held in response to great interest in the field of formal methods as a means to master the complexity of distributed systems. One of the goals of the conference was to compare and contrast various methodologies, such as constructive or property oriented. Major topics included the practical implications of formal specification techniques.

Particularly in Europe, proponents of different methods often consider themselves to be more or less competitors; investigation of possible integration, combination and unification of the various methodologies has been neglected. In the United States, on the other hand, the discussion and cooperation among representatives of different methods is considerably more prevalent.

The CONCURRENCY 88 addressed the following topics:

- Specification Languages
- Models for Distributed Systems
- Verification and Validation
- Knowledge Based Protocol Modelling
- Fault Tolerance
- Distributed Databases.

The presented papers include 12 invited, and 14 selected by the program committee. Contributions were presented by authors from Austria, the Federal Republic of Germany, France, Israel, Italy, the Netherlands, the United Kingdom and the United States.

Hamburg, August 1988

F. Vogt

Program Committee

H. Barringer (U Manchester)

M. Broy (U Passau)

D. Hogrefe (U Hamburg)

B. Mahr (TU Berlin)

G. Roucairol (Louveciennes)

R. Schwartz (Belmont)

R. Valk (U Hamburg)

F. Vogt (U Hamburg), *chairman*

Contents

Selected Papers 161

Invited Papers

While Waiting for the Millennium: Formal Specification and Verification of Concurrent Systems Now

L. Lamport
DEC Systems Research Center
Palo Alto (USA)

Abstract

Formalisms abound -- algebraic methods, CCS, I/O automata, temporal logic, ... Their proponents assure us that some day they'll be ready for real-world use. Meanwhile, industry relies on specification methods that provide syntax without semantics, and rigorous verification is for Ph.D. theses only.

We can do better. A sensible formal method, applied with common sense and realistic expectations, can help today in the design and implementation of real systems. We can't specify every relevant property of a system. We can verify only isolated, critical parts of a system, to prove the correctness only of high-level algorithms, not of executable code. Your favorite theory may remove these limitations -- some day. But tomorrow's panacea doesn't solve today's problems.

A Framework for the Synthesis of Reactive Modules

Amir Pnueli and Roni Rosner*
Department of Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

Extended Abstract

July 1988

Abstract

We consider the synthesis of a reactive module with input x and output y, which is specified by the linear temporal formula $\varphi(x, y)$. We show that there exists a program satisfying φ iff the branching time formula $(\forall x)(\exists y)A\varphi(x, y)$ is valid over all tree models.

*The work of this author was partially supported by the Eshkol Foundation.

1 Introduction

An interesting and fruitful approach to the systematic construction of a program from its formal specification is based on the idea of *program synthesis* as a theorem proving activity. In this approach, a program with input x and output y, specified by the formula $\varphi(x, y)$, is constructed as a by-product of proving the theorem $(\forall x)(\exists y)\varphi(x, y)$. The specification $\varphi(x, y)$ characterizes the expected relation between the input x presented to the program and the output y computed by the program. For example, the specification for a root extracting program may be presented by the formula $|x - y^2| < \epsilon$.

This approach, which may be called the *AE-paradigm*, or alternately, the *Skolem paradigm*, is based on the observation that the formula $(\forall x)(\exists y)\varphi(x, y)$ is equivalent to the second order formula $(\exists f)(\forall x)\varphi(x, f(x))$, stating the existence of a function f, such that $\varphi(x, f(x))$ holds for every x. If we restrict the proof rules, by which the synthesis formula is to be established, to a particular set of *constructive* rules, then any proof of its validity necessarily identifies a constructive version of the function f, from which a program that satisfies the specification φ can be constructed.

The AE-paradigm for the synthesis of sequential programs has been introduced in [WL69] (but see also [Elg61]), and has been the subject of extensive research [MW80, Con85] directed at extending the class of programs that can be synthesized, and the theories that may be used for the proofs, as well as strengthening the proof rules and the mechanisms for extracting the program from the proof.

The success of this approach to sequential programming should not be judged only by the number and complexity of programs that can be fully automatically derived, even though serious efforts are continuously invested in extending the range and capabilities of fully automatic synthesizers, in much the same way we keep improving the power of automatic theorem provers. The more important contribution of this approach is in providing a conceptual framework for the rigorous derivation of a program from its specification. Once this scheme is accepted, it can, in principle, be followed in a completely manual fashion, but encourages, on the other hand, the open ended development of a support system that will offer automatic support to increasingly larger parts of the procedure. Equally important is the realization of the identity between the processes of theorem proving and program construction. It

has been recognized very early that every system for the formal development of programs must contain at least a powerful theorem prover as an important component. The approach of synthesis by theorem proving tells us that such a system need not contain much more than a theorem prover.

In view of the success of this approach for sequential programs, there is no wonder that several attempts have been made to extend it to concurrent programs. These attempts were held back for awhile by the question of what was the appropriate language to use for expressing the specification formula φ. While, for sequential programs, it is obvious that a properly enriched first order language is adequate, it took time to propose similarly adequate specification language for concurrent programs. One of the more stable proposals is that of *temporal logic* ([Pnu77, GPSS80, MP82, SC85, Pnu86]). The basic supposition underlying temporal logic is that concurrent programs often implement *reactive systems* (see [HP85, Pnu85]) whose role is not to produce an output on termination, but rather to maintain an ongoing interaction with their environment. Therefore, the specification should describe the expected *behavior* of the system throughout its activity.

Indeed, the two main works on the synthesis of concurrent programs, which are reported in [CE81] and [MW84], consider a temporal specification φ, and show that if it is satisfiable, we can use the model that satisfies φ to construct a program that implements φ.

There are, however, some limitations of the approach, as represented in these two pioneering contributions, due to the fact that the approach is based on *satisfiability* of the formula expressing the specification $\varphi(x, y)$. The implied limitations are that the approach can in principle synthesize only *entire* or *closed* systems.

To see that, assume that the system to be constructed has two components, C_1 and C_2. Assume that only C_1 can modify x (a shared variable used for communication) and only C_2 can modify y. The fact that $\varphi(x, y)$ is satisfiable means that there exists at least one behavior, listing the running values of x and y, which satisfies $\varphi(x, y)$. This shows that there is a way for C_1 and C_2 to cooperate in order to achieve φ. The hidden assumption is that we have the power to construct both C_1 and C_2 in a way that will ensure the needed cooperation. If indeed we are constructing a closed system consisting solely of C_1 and C_2 and having no additional external interaction, this is quite satisfactory.

On the other hand, in a situation typical to an *open* system, C_1 represents

the *environment* over which the implementor has no control, while C_2 is the body of the system itself, to which we may refer as a *reactive module*. Now the situation is no longer that of peaceful cooperation. Rather, the situation very much resembles a two-person game. The module C_2 does its best, by manipulating y, to maintain $\varphi(x,y)$, despite all the possible x values the environment keeps feeding it. The environment, represented by C_1, does its worst to foil the attempts of C_2. Of course, this anthropomorphism should not be taken too literally. The main point is that we have to show that C_2 has a winning strategy for y against all possible x scenarios the environment may present to it.

It seems that the natural way to express the existence of a winning strategy for C_2, is again expressed by the AE-formula $(\forall x)(\exists y)\varphi(x,y)$. The only difference is that now we should interpret it over temporal logic, where x and y are no longer simple variables, but rather sequences of values assumed by the variables x and y over the computation. In contrast, we may describe the approach presented in [MW84] and [CE81] as based on the formula $(\exists x)(\exists y)\varphi(x,y)$.

This is indeed the main claim of this paper. Namely, that the theorem proving approach to the synthesis of a reactive module should be based on proving the validity of an AE-formula. As we will show below, the precise form of the formula claiming the existence of a program satisfying the linear time temporal formula $\varphi(x,y)$, is $(\forall x)(\exists y)A\varphi(x,y)$, where A is the "for all paths" quantifier of branching time logic. Thus, even though the specification $\varphi(x,y)$ is given in linear logic, which is generally considered adequate for reactive specifications, the synthesis problem has to be solved in a branching framework. This conclusion applies to the synthesis of both synchronous and asynchronous programs, yet for simplicity of presentation, we prefer to restrict the exposition in this paper to the synthesis of synchronous programs. The application of our approach to the synthesis of asynchronous programs will be presented in a subsequent paper.

An interesting observation is that the explicit quantification over the *dynamic* (i.e., variables that may change their values over the computation) interface variables, x and y, is not absolutely necessary. As we will show in the paper, there exists an equivalent branching time formula, which quantifies only over *static* variables (i.e., variables which remain constant over the computation), whose *satisfiability* guarantees the existence of a program for $\varphi(x,y)$. For the case of finite state programs, this other formula be-

comes purely propositional, and its satisfiability, therefore, can be resolved by known decision methods for satisfiability of propositional branching time formulae. However, for the more general case that deductive techniques have to be applied, we prefer to establish validity, rather than satisfiability, in particular since the explicitly quantified version emphasizes the asymmetry between the roles of the variables x and y in the program.

We justify our main claim on two levels. First we consider the general case and show that the synthesis formula is valid iff there exists a *strategy tree* for the process controlling y. We then argue that such a strategy tree represents a program by specifying an appropriate y for each history of x values. On this level we pay no attention to the question of how effective this representation of a program is, which becomes relevant when we wish to obtain a program represented in a conventional programming language.

Hence, in a following section we consider the more restricted case in which the specification refers only to *Boolean variables*. In this case the validity of the synthesis formula is *decidable*, and we present an algorithm for checking its validity and extracting a finite-state program out of a valid synthesis formula.

A related investigation of synthesis for the finite state case, based on a similar approach, has been carried out in [BL69]. The question, formulated for the first time in [Chu63], has been asked in an automata-theoretic framework, where the specification $\varphi(x, y)$ is given by an ω-automaton accepting a combined x,y-behavior, and the extracted automaton is an ω-transducer. The solution presented in [BL69] uses game-theoretic ideas, and it is of very high computational complexity. Later, [HR72] and [Rab72] have observed, similar to us, that even though the specification is expressed by automata on strings (corresponding to linear temporal logic), its synthesis must be carried out by using tree automata. In our own approach we had to use a similar algorithm for checking emptiness of ω-tree automata. In [PR88], we show how the complete synthesis process can be performed in deterministic time which is doubly exponential in the size of the specification.

2 Temporal Logic

We describe the syntax and semantics of a general branching time temporal language. This language is an extension of CTL* ([CES86, EH86, HT87]),

obtained by admitting variables, terms, and quantification. Its *vocabulary* consists of *variables* and *operators*. For each integer $k \geq 0$, we have a countable set of k-ary variables for each of the following types: *static function variables* — F^k, *static predicate variables* — U^k, *dynamic function variables* — f^k, and *dynamic predicate variables* — u^k. The intended difference between the dynamic and the static entities is that, while the interpretation of a dynamic element in a model may vary from state to state, the interpretation of a static element is uniform over the whole model. For simplicity, we refer to 0-ary function variables simply as (individual) variables, of which we have both the static and the dynamic types. The operators include the classical $\neg$, $\vee$, $\exists$ and the temporal $\bigcirc$ (next), $\mathcal{U}$ (until) and E (for some path).
Terms: For every $k \geq 0$, if $t_1, \cdots, t_k$ are all terms, then so are $F^k(t_1, \cdots, t_k)$ and $f^k(t_1, \cdots, t_k)$.
State formulae are the (only) formulae defined by the following rules:

1. For all $k \geq 0$, if $t_1, \cdots, t_k$, are all terms, then $U^k(t_1, \cdots, t_k)$ and $u^k(t_1, \cdots, t_k)$ are (atomic) state formulae.

2. If p and q are state formulae, then so are $\neg p$, $(p \vee q)$ and $(\exists \alpha)p$ where α is any variable.

3. If p is a path formula then Ep is a state formula.

Path formulae are the (only) formulae defined by the following rules:

1. Every state formula is a path formula.

2. If p and q are path formulae, then so are $\neg p$, $(p \vee q)$, $\bigcirc p$ and $(p \mathcal{U} q)$.

We shall omit the superscripts denoting arities and the parentheses whenever no confusion can occur. We also use the following standard abbreviations: T for $p \vee \neg p$, F for $\neg$T, $p \wedge q$ for $\neg(\neg p \vee \neg q)$, $p \rightarrow q$ for $\neg p \vee q$, $p \equiv q$ for $(p \rightarrow q) \wedge (q \rightarrow p)$, $(\forall \alpha)p$ for $\neg(\exists \alpha)(\neg p)$, $\Diamond p$ for T$\mathcal{U}p$, $\Box p$ for $\neg \Diamond \neg p$, pUq for $p \mathcal{U} q \vee \Box q$ and Ap for $\neg E(\neg p)$. We shall use the letters a, b for static individual variables (*constants*) and x, y for dynamic individual variables, or *propositions*, for the Boolean case.

By restricting this syntax, one gets special cases of the temporal language. In *classical static* logic, there are no dynamic variables nor temporal operators $(\bigcirc, \mathcal{U}, E)$. *First-order* logic permits $\exists$ quantification over individual

variables only. In *propositional* (resp. *quantified propositional*) logic, the only variables permitted are propositions, without (resp. *with*) $\exists$ quantification over them. Finally, *linear* temporal logic omits the E operator (i.e. all linear formulae are path formulae).

The semantics of temporal logic is given with respect to *models* of the form $\mathcal{M} = \langle D, I, M \rangle$. D is some non-empty data *domain*, I is a *(static) interpretation* of all static variables into appropriate functions and predicates over D, and $M = \langle S, R, L \rangle$ is a *structure*. S is a countable set of *states*. $R \subseteq S \times S$ is a binary total *access relation* on S. L is the *labeling function*, assigning to each state $s \in S$ a *(dynamic) interpretation* $L(s)$ of all dynamic variables into appropriate functions and predicates over D. A *path* in M is a sequence $\pi = (s_0, s_1, \cdots)$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. A *fullpath* in M is an infinite path. Denote by $\pi^{(j)} = (s_j, s_{j+1}, \cdots)$ the j^{th} *suffix of* π.

A structure $M = \langle S, R, L \rangle$ is called a *tree-structure*, **iff** the following conditions are satisfied:

1. There exists a precisely one state, $r \in S$, called the *root* of M, which has no *parent*, i.e., no state $s \in S$, such that $R(s, r)$.

2. Every other state $t \neq r$, has precisely one parent.

3. For every state $s \in S$, there exists a unique path leading from r to s.

A model $\mathcal{M} = \langle D, I, M \rangle$, is called a *tree-model*, **iff** the structure M is a tree-structure.

In order to interpret applications of static functions and predicates to terms, and applications of existential quantifiers over static variables to formulae, we use standard definitions, over the static interpretation I and the semantics of the given terms or formulae. Analogously, applications of dynamic functions are interpreted with respect to dynamic interpretations.

Satisfiability of a state formula is defined with respect to a model $\mathcal{M}$ and a state $s \in S$, inductively as follows:

1. $\langle \mathcal{M}, s \rangle \models u(t_1, \cdots, t_k)$ **iff** $L(s)(u)(d_1, \cdots, d_k) = true$, where d_i is the semantics of t_i in $\langle \mathcal{M}, s \rangle$, $1 \leq i \leq k$. That is, we apply u, as interpreted in s by L, to the evaluation of $t_1, \cdots, t_k$, as interpreted in s.

2. $\langle \mathcal{M}, s \rangle \models \neg p$ **iff** not $\langle \mathcal{M}, s \rangle \models p$.