Compositional Verification of Continuous-Discrete Systems

Ralf Huuck[1], Ben Lukoschus[1], Goran Frehse[2], and Sebastian Engell[2]

[1] University of Kiel, Institute of Computer Science and Applied Mathematics, Chair of Software Technology, D-24098 Kiel, Germany, {rhu,bls}@informatik.uni-kiel.de
[2] University of Dortmund, Process Control Lab (CT-AST), D-44221 Dortmund, Germany, {g.frehse,s.engell}@ct.uni-dortmund.de

Abstract. Hybrid systems are well-suited as a design and modeling framework to describe the interaction of discrete controllers with a continuous environment. However, the systems described are often complex and so are the resulting models. Therefore, a formal framework and a formal verification to prove the correctness of system properties is highly desirable. Since complexity is inherent, standard formal verification techniques like *model checking* soon reach their limits. In this work we present several options how to tackle the complexity arising in the formal verification of hybrid systems. In particular we combine the model checking approach with *abstraction* and *decomposition* techniques such as the *assumption/commitment* method as well as deductive methods.

1 Introduction

The description of real-world physical systems has always been an issue. Such a system model not only enhances the understanding of the underlying physics but it makes it possible to actually predict the system's behavior. Since nowadays nearly every production, power generation, and logistics process is highly automated, such a prediction is extremely valuable in order to simulate the system's behavior in different environments or even to prove that certain properties are satisfied.

However, every system model is by nature an abstraction of the real world. Finding the right abstraction and, thus, developing the right system model is not an easy task. In general the abstraction and, therefore, the model is chosen according to the design level one is interested in. At times the detailed continuous physical behavior is the focus of study and other times discrete behaviors such as communication and synchronization is of main interest.

Continuous models are used, e.g., to describe movements of mechanical systems, linear circuits or chemical reactions while discrete models are sufficient to describe the collision of two objects in a mechanical system, the switching in circuits or the use of pumps and valves in chemical plants. Continuous models are generally given in the form of differential equations, possibly supplemented by a set of algebraic constraints. In contrast, discrete models are more diverse but often can be captured by some form of a state representation. Physical processes are controlled by software on digital computers. Such embedded control systems combine continuous physical behavior with discrete control algorithms and are called *hybrid systems.*

To model and to verify hybrid systems which describe the interaction of control software with a physical plant is a very desirable goal. However, it is not easy to achieve since it requires a unified framework covering the continuous and discrete world and a verification framework which is able to cope with such systems. This is even more difficult when thinking of exhaustive verification that is as much automated as possible. This means, from a formal system description and a set of requirements the verification task can be left to a computer. Despite theoretical limitations of this approach any verification technique has to cope with the immense complexity that lies in the nature of such a system.

Then, why to use formal description methods and formal verification at all? In the current design system processes some methods to enhance the quality of the software and the overall system have already made their way into industrial standard practice. These comprise techniques such as listing software requirements and striving for a clean documentation for the design process as well as code review, system simulation and testing to check for the correctness of the implemented system.

However, techniques in common practice have various drawbacks.

- They often lack a formal basis. For instance, the specification is defined in natural language, which easily leads to misunderstanding and misinterpretation.
- The requirements are not complete, i.e., there are cases which are not taken into account. Hence, parts of the system remain unspecified and, thus, are allowed to behave differently than the designer had in mind. Especially without a formal model it is more likely to forget cases, or cases which are defined in a contradictory way remain undetected.
- The verification of the implementation might be approached in an unorganized way. For instance, testing is done with some arbitrary inputs when it is more efficient to choose data according to the boundary conditions of the model or the implementation, simulation focuses on “non-important” variables and code review neglects inter-procedural dependencies etc.
- Most informal techniques like testing or simulation are not exhaustive, i.e., the they do not cover all program executions and thus give way to subtle but often fatal flaws.

Formal methods promise to remedy the above mentioned weaknesses. However, formal methods are not fool-proof by themselves. Sometimes they require exhaustive knowledge in mathematics, logics and the understanding of the system. Moreover, they do not a priori prevent forgetting about requirements or even ensure to appropriately map the real world to the model. Especially in software design one often has a clear idea of what to achieve, but much less of how to specify this formally or even to define what is considered to be legal and what to be harmful. Formal methods provide tools to further investigate into the design and verification process and allow to enhance the quality of the system significantly, but they do not buy any guarantee that what you prove is what you have in mind.

This work concentrates on applying formal methods to hybrid systems while at the same time tackling the inherent complexity issues. The approach presented here does not take the whole system at once into account, but divides the system and, hence, the verification task, into several components and several layers of abstractions. The division and verification of single components is known as *compositional verification*. However, hybrid systems are often too complex and too interwoven to find single components that can be verified independently of the remaining systems. It is much more natural to make some *assumptions* about the behavior of the remaining system under which the selected component can be verified. The verification of the component results into some *commitment* this unit fulfills under the given assumptions. Applying this method to every component still leaves the task to combine the assumptions and commitments in a meaningful and non-contradictive way. In this work we present a framework which allows to reason in this so-called *assumption/commitment style* and supports a formal and automated verification as far as possible.

We start with a brief overview on software verification and compositional techniques in Section 1.1. In Section 2 an introduction into a standard formal verification technique called *model checking* (11, 51) is given. We explain the basic terms as well as logics and techniques used. The subsequent Section 3 deals with complexity issues of formal verification and hybrid systems.

1.1 Historical Notes

This section gives a brief summary of the development of formal software verification approaches in general and some important compositional methods in particular. More extensive surveys can be found in (14, 15).

Formal Verification of Software From the beginnings of the computer age, verification of software has always been an issue for programmers and system developers. Pioneers like John von Neumann and Alan Turing already thought about the correctness of programs for the first computers (22, 56).

In 1967 Robert W. Floyd presented the *inductive assertion method* (19), a formal strategy to prove the correctness of sequential programs written as labeled transition systems. A similar approach was presented by Peter Naur (45) one year earlier. C.A.R. Hoare axiomatized this method into a compositional style for sequential programs (26). Programs are annotated with assertions, and their correctness is proven locally. Then the local assertions can be combined in a compositional fashion to obtain a global specification.

This Hoare-style proof system was extended to concurrent shared-variable programs in 1976 by Susan S. Owicki and David Gries (47). Their method however involves a so-called *interference-freedom test,* which operates on every combination of local control locations and therefore is non-compositional.

Proof systems for programs with distributed synchronous communication were independently developed by Krzysztof R. Apt, Nissim Francez and Willem-Paul de Roever (3) and by Gary M. Levin and David Gries (37). Here a so-called *cooperation test* is done for every combination of input and output actions, which is also non-compositional.

In 1977 Amir Pnueli developed the temporal logic approach for the verification of concurrent programs (48), for which he received the Turing Award in 1996. This method is also non-compositional.

Compositional Approaches In 1969 Edsger W. Dijkstra was the first to publish within the computer science community the opinion that compositional reasoning is needed for the formal verification of large programs (16).

Cliff B. Jones developed a compositional verification approach for concurrent shared-variable programs (32, 33). His so-called *rely-guarantee* formalism, which specifies a system by its desired properties (guarantee) provided that its environment behaves in a certain way (rely).

A similar compositional approach for distributed synchronous communication, called the *assumption-commitment* method, was presented by Jayadev Misra and K. Mani Chandy in 1981 (44).

Within the field of process algebra – the main languages used include CCS (42, 43), CSP (8, 27), and ACP (5) – one has always been striving for compositional reasoning, e.g., by defining behavioral preorders which are preserved by the composition operators.

2 Model Checking

Common to every verification task is to prove that a system, a program or simply an abstract model of a problem satisfies certain requirements. Formally, this is denoted by

$$M \models \varphi,$$

where M is a model of the system, φ is the requirement and $\models$ denotes the satisfaction relation. model checking (11, 51) is an algorithmic way to decide whether M satisfies φ. Although any verification approach is based on this, the actual logic or – more general – the formalism to denote these three items varies a lot. In the following we present some formal models for each of them. We mainly focus on the ones we will use throughout this work.

2.1 System Model

In this article we concentrate on verification of *reactive systems.* These are systems which communicate with their environment and may often – like operating systems – not terminate. Hence, a model which captures their infinite behavior in a concise way is desirable. Simply specifying their input/output behavior is not sufficient, it is rather interesting to know the *states* of a system, too.

Therefore, we start by describing the behavior of a system with some state-based formalism. Such formalisms include Petri nets (52), CSP (8, 27), CCS (42, 43), different forms of automata, LOTOS (7), SDL (53), etc. In these formalisms the behavior of the system is described in terms of local state changes or events. The global behavior of the system is given as the state-space generated from the system description.

In this work we use different kinds of automata for the system description, namely discrete, timed and hybrid automata.

Discrete Automaton A discrete automaton $A = (Q, q_0, \delta, F)$ over an alphabet Σ (events, actions) is a structure where

- Q is a finite set of control locations,
- q_0 is an initial location,
- $\delta : Q \times \Sigma \longrightarrow Q$ is a transition function, and
- F is an acceptance condition.

A sequence of actions in Σ which is produced by taking a path through the automaton, starting with the initial location and satisfying the acceptance condition, is called a *word.* The set of all words, i.e., the set of all possible sequences, for an automaton A is called the *language* of A denoted by $L(A)$. The acceptance condition can vary from a single location which indicates the end of the sequence once it is reached to a set of locations which have to be reached infinitely often. The acceptance condition mainly determines the different kinds of discrete automata which can be found in the literature.

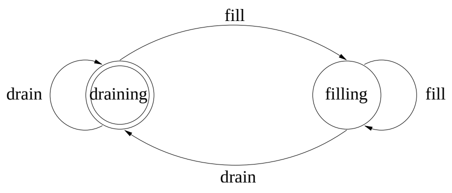

Fig. 1. Discrete automaton for a tank model

An example for a discrete automaton is depicted in Figure 1. This automaton gives a rough model for a tank. There are two control locations *draining* and *filling*. The double framed circle around *draining* indicates the initial location. Depending on the actions *drain* and *fill* transitions depicted by arrows are taken. We do not give an explicit acceptance condition here and note that the model is very simplified, i.e., it is not captured that the tank might run empty or might overflow.

Timed Automaton In contrast to discrete automata, the setting of timed automata is in a dense real-time world. To express quantitative time, *clocks* are introduced which are real-valued variables evolving over time. Moreover, they can be checked against thresholds, and they can be reset when a transition is taken.

Formally, a timed automaton over an alphabet Σ is a quadruple $T = (Q, q_0, C, E)$ where

- Q is a finite set of locations,
- q_0 is the initial location,
- C is a finite set of clocks, and
- E is a set of edges of the form (q, γ, a, ρ, q'), where $q, q' \in Q$ are the source and target locations, γ is a *transition condition*, i.e., a Boolean formula over clock variables and thresholds, $a \in \Sigma$ is an action and ρ is the set of clocks that are reset when taking this transition.

The language of a timed automata is given by the set of all execution sequences over time. Traditionally, only infinite sequences are considered.

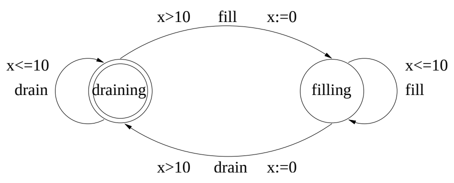

Fig. 2. Timed automaton for a tank model

A timed automaton example is depicted in Figure 2. In contrast to Figure 1 there is a clock x which constraints the moments transitions are taken. This clock serves as a timer for draining and filling periods. Starting from the initial location *draining* the location is changed only if x is greater than 10. If so the x is reset and control will reside in location *filling* until the clock value exceeds 10 again. In the meanwhile self-loops are possible.

Hybrid Automaton Discrete automata do not incorporate quantitative time, but timed automata do so by the use of clocks. However, they do not model arbitrary continuous functions. This feature is covered by so-called hybrid automata. These allow do model and reason about a set of continuous variables evolving over time.

Formally, a hybrid automaton $H = (Q, q_0, Var, E, Act, Inv)$ over an alphabet Σ consists of

- a finite set of locations Q with some initial location q_0,
- a finite set of real-valued variables *Var*.
- a finite set E of discrete transitions. Each transition $e = (q, \rho, a, q')$ between two locations $q, q' \in Q$ labeled by some action $a \in \Sigma$ depends on a transition condition ρ which reasons about the variables in *Var*,
- a labeling function *Act* that assigns a set of activities to each location $q \in Q$. The activities describe how the variables in *Var* evolve continuously as long as control resides in q.

The semantics of an hybrid automaton is defined by all trajectories of the continuous variables as well as the actions over time.

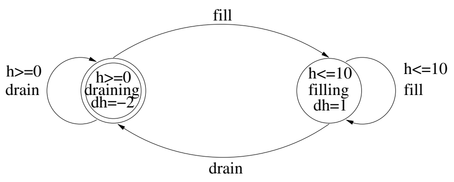

Fig. 3. Hybrid automaton for a tank model

A hybrid model for the tank example is shown in Figure 3. This model describes the tank level h in a filling and draining process. Draining is two times faster than filling. Although possible, there are no guards or resets on the transitions, but invariants in the locations determine when exactly control is allowed to stay there. Note that there also exist different timed automaton models with invariants, deadlines and urgency. Mostly this has little effect on the expressiveness (cf. (54)), but allows more or less convenient notations.

2.2 Computational Model

In the previous section we described very briefly how to derive a behavior from each description model. However, the models themselves are merely syntax and in order to formally derive a *semantics*, i.e., the system's behavior, the system description can be mapped to a mathematical abstract representation. This abstract representation is also called *computational model* and represents the semantics.

One way to describe a computational model is a state transition system. It consists of states and and has also the ability to represent the fact that in any given state the system reacts to certain actions and might enter new system states. This pair of system states is then called a *transition*. The semantics of a system is then determined by the sequences of all transitions in a system that start from some given initial state. One formal way to describe these state transition systems are *Kripke structures* (34), named after the logician Saul A. Kripke who used transition systems to define the semantics of modal logics. Transition systems are graphs consisting of states, transitions and a function that maps each state to a set of properties which hold in that state.

Formally, we define a Kripke structure as follows: Given a set of atomic properties P, also called propositions, a Kripke structure $K = (S, S_0, R, \mu)$ contains the following components:

- S is a set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a transition relation, which is required to be total, i.e., for every state $s \in S$ there exists an $s' \in S$ such that $(s, s') \in S$, and
- $\mu : S \longrightarrow 2^P$ is a labeling function that assigns a set of propositions to every state.

An execution sequence of a Kripke structure is defined as a possibly infinite sequence $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 \in S_0$ and for every index $i > 0$ in π we have

$(s_{i-1}, s_i) \in R$. This means starting from the initial state we go along a path in the graph represented by the Kripke structure. The semantics of a system described by a Kripke structure is the set of all its sequences, i.e., all possible paths from all initial states.

In order to describe the semantics of a system model it is translated into such a computational model first. This means, the system model represents the syntax and the computational model the semantics. For the different types of automata presented above, the computational models are also different. While discrete automata only have to reflect the control location in a state, timed and in particular hybrid systems need to reflect time as well in a state. Since time is dense for both the latter models, it is not always guaranteed to find a finite representation of these systems. However, using abstract or symbolic state representations, i.e., the clustering of concrete states into equivalence classes, in many cases a finite representation is possible also for timed and so called *linear hybrid automata*. The latter are hybrid automata which only allow fixed (but arbitrary) rates for the continuous variables. A finite representation is important in order to guarantee termination for algorithmic approaches like model checking.

2.3 Temporal Logics

Describing the system formally is only one thing. For verification it is also necessary to describe the requirements posed to a system in a formal style. There are different ways to do so. One fundamental issue is to choose between an operational or a declarative way. In this context operational means, e.g., using automata itself in order to specify the desired properties. The advantage is that the same framework for system modeling is also used to specify the system requirements. However, it is often a bit tedious to formulate requirements as automata, and automata are sometimes not as easy to understand as requirements. The declarative way means using logics to specify the requirements.

As mentioned before, we are mainly interested in reactive systems and, therefore, are concerned about the states of a system as well as the transitions between these states. Since basic propositional logic allows to reason about states only but not sequences of states or transitions, so-called *temporal logic* (49) is used in order to remedy this fact. Temporal logic extends propositional logic, i.e., Boolean proposition with connectivities such as logical conjunction, disjunction and negation, with *modal operators*. These are operators like *always* or *eventually* that allow reasoning over execution sequences and can be combined with the usual connectivities.

Let us define propositional logic first. Based on propositions p logical expressions can be constructed by the following rules:

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$$

Other Boolean connectives like “$\vee$”, “$\Rightarrow$”, and “$\Leftrightarrow$” can be derived from “$\neg$” and “$\wedge$” as usual.

The semantics is straightforward and we is not shown here. Next, we present the extension from propositional to temporal logic. In general we can define and distinguish between two main temporal sub-logics, namely, linear time and branching time.

Linear Time Temporal Logic One way to describe requirements is to define desired sequences in time. Linear Time Temporal Logic (LTL) allows to reason about paths in computational models like Kripke structures. In order to do so, propositional logic is extended by the following basic modal operators:

- $\bigcirc$. This denotes the modality "next" and requires that a property holds in the next state of a path, e.g., a path π in a Kripke structure satisfies $\bigcirc\varphi$ if and only if φ is satisfied in the second state of π.
- $\mathcal{U}$. This denotes the infix modality "until". I.e., a path π in a Kripke structure satisfies the expression $\varphi\mathcal{U}\psi$ if and only if ψ is satisfied in some later state of π, and φ holds in all states in between, including the first state of π. This is meant by the expression "φ until ψ".

LTL is founded on these basic modalities and their free combination with propositional logic. From these the following useful abbreviations can be defined:

- $\Diamond$ means "eventually", and a path π satisfies the expression $\Diamond\varphi$ if and only if there exists a state in π which satisfies φ.
- $\Box$ means "always", and a path π satisfies $\Box\varphi$ if and only if all states in π satisfy φ.

Branching Time Temporal Logic In contrast to LTL branching time logics do not reason over single paths but over sets of paths, more precise, trees. One logic which does so is called Computational Tree Logic (CTL) which is propositional logic extended by *path quantifiers* and *temporal operators*. The temporal operators are the same as in LTL presented above. The path quantifiers are "$\exists$" which requires a single path to exist that satisfies some property and "$\forall$" which requires all paths of the computational model to satisfy some property.

CTL formulas are constructed from propositional logic, temporal operators and path quantifiers in the following way: Every formula starts with a path quantifier, every path quantifier is immediately followed by a temporal operator, and every temporal operator is preceded by a path quantifier.

This allows to build formulas such as

- $\exists\Box\varphi$, which means that there exists a path where always, i.e, for all states, φ holds, and
- $\exists\Diamond\forall\Box\varphi$, which means there exist a path with a certain state from whereon for all paths, i.e., branches, φ is always true.

Remarks Note that CTL and LTL not only use different means to describe system properties, but in general there are LTL formulas which cannot represented in the CTL framework, and vice versa. Moreover, while linear time appears to be conceptually simpler than branching time, the latter is often computationally more efficient.

For both types of logics there exist real-time extensions. This means the logics provide the possibility to reason about explicit time and distances. We do not go into detail here.

2.4 Tools and Limitations

Returning to the initial task of checking $M \models \varphi$, model checking is, as mentioned, an algorithmic (i.e., automatic) way to decide whether a model M satisfies φ or not. There are several tools supporting model checking. For discrete automata and logics like CTL or LTL there are SMV (40) and SPIN (28) as the most prominent ones. For checking timed automata with real-timed logics there are UPPAAL (36), KRONOS (46) and extensions of SPIN. For linear hybrid systems HyTech (23) is a tool that enables to check reachability of certain states of the corresponding linear hybrid automaton. Moreover, there are many more tools which are also based on other system description models as well as logics.

For checking reactive systems one of the presented system models and logics is often used. However, due to fundamental limitations not every model and every logics is applicable for model checking. Timed and even more hybrid systems are restricted to certain classes, since a finite state representation in whatever way has to be guaranteed in order to keep model checking possible. Problem classes for which there cannot be any general algorithmic solutions are called *undecidable*.

Despite of these basic fundamental restrictions model checking has also to cope with serious complexity issues which are described in the next section.

3 Complexity Issues

One of the main drawbacks of state-based formal verification methods is the so-called *state explosion problem:* When a large system consists of several smaller components (e.g., automata) running in parallel, the number of global states increases exponentially with the number of components. For instance, consider a system of 20 automata working in parallel, each of which having 10 local states. This amounts to 10^{20} global states. The simple task of enumerating these states on a machine that needs only one nanosecond per state (which is considerably fast at the time of writing) already takes over 3000 years. Building and searching a graph based on these states takes significantly longer and is far beyond today's memory capabilities.

The state explosion problem is inherent in any system having parallel structures and poses a major complexity problem to any verification method based on the exhaustive enumeration of global states. Several techniques have been developed to minimize the impact of this problem on the time and memory consumption of the model checking process. Often a model checking algorithm uses a combination of several such techniques, which are discussed in the following.

Note that although all these methods can result in a significant speedup in practice, they are limited by the worst case complexity inherent to the problem (35, 38, 57). E.g., model checking LTL or CTL properties for Kripke structures is polynomial in $n \log m$ where n is the length of the formula and m is the size of the Kripke structure. When it comes to concurrent programs, i.e, different automata composed in parallel the problem is already PSPACE-complete even for a fixed formula. The same holds for model checking real-time systems in a timed variant of CTL.

3.1 Global vs. Local Strategy

In accordance with the two parameters of the model checking problem, the model M and the requirement φ, there are two basic strategies when designing a model checking algorithm, the “global” and the “local” strategy (41). “Global” means the algorithm operates recursively on the structure of φ and evaluates each sub-formula over the whole M, while the local strategy checks only parts of the state space at a time but for all sub-formulas of φ. The worst-case complexity of both approaches is the same, however, the average behavior can differ significantly in practice. Traditionally, LTL model checking is based on local approaches while for CTL global algorithms are applied.

3.2 “On-the-fly” Techniques

The classical model checking approach builds a complete state transition graph of the system and performs a search on this graph. But often a large part of the graph is not traversed during the search or is even unreachable from the initial state(s) of the search. Therefore it is often a good idea to construct the graph in an “on-the-fly” fashion (6, 13). That is, only the part of the graph that is currently needed is constructed during the search and kept in memory for later reuse, often supported by caching algorithms.

3.3 Efficient Data Structures

A considerable amount of memory can be saved using efficient data structures during the model checking process. One prominent example are *binary decision diagrams* (BDDs) (9, 10), which are used as a compact representation of Boolean functions. Ken McMillan suggested in his PhD thesis (39) to use them for model checking, and today BDDs and similar data structures are the key solution for efficient memory usage in many kinds of computation software.

In the field of timed automata the observation that despite their continuous nature, clocks are often compared only to each other and a finite and bounded number of constants, opened the possibility to discretize the state space for model checking. So called *clock regions* are stored in data structures like *difference bounded matrices* (DBMs) (4, 18) and are used in most model checking tools for timed automata like KRONOS (46) and UPPAAL (36).

3.4 Abstraction

Abstraction is a fundamental concept used in all formal verification methods. Abstracting means replacing a concrete object with an abstract one which is more universal, and therefore, often has a simpler structure than before. A well-chosen abstraction simplifies as much as possible, without losing too much information about the concrete object. Abstractions can be used in different ways during the specification and verification process:

- Building the system model: Every translation from a real-life system or an informal system description into a formal model is an abstraction.
- Optimizing the system model: Depending on the property that is to be checked, different abstractions of the system model can be useful, e.g., by abstracting from data, time, or continuous variables to obtain simpler models.
- Reducing the complexity of model checking: Model-checkers often use abstractions to minimize time and space usage, e.g., by introducing symbolic states.

When abstracting a system model, often a so-called *safe abstraction* is chosen: Whenever a property holds for the abstract system, it also holds for the concrete system. The converse, however, does not always hold, due to the over-approximation which occurs in the abstraction process. A positive model checking result on a safe abstraction therefore means that the concrete system also fulfills the property, whereas a negative result can either mean that the concrete system is not correct or that the abstraction is too coarse.

Thus, when getting a negative result, the counterexample provided by the model-checker is examined to see if the error will also occur in the concrete system. If it doesn't, a finer abstraction has to be chosen.

3.5 Compositionality

Another important concept is compositionality. In a compositional approach the system model is split into components. Each component is then specified as a single entity, and its correct behavior can be proved by model checking. The specifications of all components are then combined to get the global property of the system model. A prerequisite for this approach is that the behavior of the components is completely described by its specifications such that the behavior of the global system model only depends on these specifications and not on any additional information about the internal structure of the components.

The advantage of such an approach is obvious. Consider the example at the beginning of this section (20 automata, 10 local states each). A compositional approach yields 20 applications of a model checking algorithm, each of which involving only 10 states, whereas the global approach applies model checking once, but on a set of 10^{20} states. There is, however, some (often significant) overhead for the decomposition of the system model and the construction and the composition of the local specifications.

Section 4 discusses the compositional verification approach which is subject of our research project “Integrierte algorithmische und deduktive Verifikation verteilter Steuerungssysteme für hybride Prozesse” (“Integrated algorithmic and deductive verification of distributed control systems for hybrid processes”) in the DFG KONDISK program.

4 Compositional Verification

4.1 The Idea

A compositional approach to verification aims at deducing properties of a system from a local analysis of its constituent parts. Since each subsystem, or module, is dependent on inputs from its environment, this environment must somehow be represented to carry out a local analysis. In the trivial case the module’s behavior is unchanged by the environment, whilst in the worst case the interactions might be so intensive that any useful analysis requires a representation of the environment that is equivalent to the composed system.

However, in some domains of application, such as chemical engineering, the modules depend only on a few other modules and only via a few interface channels. In that case, a simplified representation of the environment will enable a less complex local analysis. The problem is how to:

- obtain such a simplified representation and
- ensure that the local analyses do indeed allow deductions about the composed system.

One approach is to compose the environment of a module and then simplify it step by step. This can be referred to as *compositional minimization*. The simplification method must ensure the validity of the deduction, i.e., conserve certain properties with respect to composition.

In the next section some notation is introduced, afterwards the assumption/ commitment methods is presented followed by the formulation of two proof rule paradigms. Finally, the approach is illustrated by an example.

4.2 Groundwork

Modules and Environments Consider a system S that can be divided into several *modules*, or subsystems, working in parallel:

$$S = S_1 || \dots || S_n. \tag{1}$$

The respective *environment* E_i for each module S_i is the composition of the remaining automata of S:

$$E_i = S_1 || \dots || S_{i-1} || S_{i+1} || \dots || S_n. \tag{2}$$

The behavior of a module can be represented by a discrete or hybrid automaton S. In order to specify that a module fulfills certain requirements, two formalisms exist: properties and abstractions.

Properties A property of an automaton can be specified in a temporal logic formula. This provides a compact description of a requirement if it concerns only a certain aspect of the behavior of the automaton. However, formulas can become very long and tedious to handle manually.

Abstractions If a requirement defines the set of desired behavior in an exhaustive manner, it may better be described by an automaton. In practice, the desired behavior of an automaton S_i can be specified as $\hat{S}_i$ by copying the automata while omitting all undesired locations and states. The abstraction is denoted as $S_i \preceq \hat{S}_i$, meaning that any behavior of S_i finds a matching representation within the specification $\hat{S}_i$.

Tableaux and Test Automata A subclass of temporal logic formulas, sufficiently large for practical applications, can also be represented by automata (12). The automaton T_ϕ representing a formula ϕ can be derived algorithmically by a *tableaux* construction. As a result, a *test automaton* $\hat{S}_i^T$ can be constructed in order to verify an abstraction $S_i \preceq \hat{S}_i$ using model checking. The test automaton contains a *fail* state that is reachable if $S_i \not\preceq \hat{S}_i$ so that

$$S_i||\hat{S}_i^T \models \neg reach(fail) \quad \Rightarrow \quad S_i \preceq \hat{S}. \tag{3}$$

4.3 The Assumption/Commitment Paradigm

Consider the behavior of a module S_i. Let

$$S_i \models (a_i, c_i), \tag{4}$$

denote that S_i commits itself to fulfilling the *commitment* c_i under the *assumption* a_i. The pair (a_i, c_i) is called an *assumption/commitment-pair* (a/c-pair). A number of alternative notations can be found in literature, e.g., $\langle a_i \rangle S_i \langle c_i \rangle$ (50).

The goal of the compositional analysis is to show that the composed system S fulfills a certain requirement corresponding to a global commitment c. As an a/c-pair, this is written as $S \models (true, c)$. If S is a system that depends on outside input, e.g., human interaction, additional global assumptions a about the unspecified environment of S can be included:

$$S \models (a, c). \tag{5}$$

The a/c-method consists of finding local a/c-pairs (a_i, c_i) for each module S_i such that the combination of the commitments fulfills the assumptions in such a way that the conclusion (5) holds. A major problem results from the fact that if the a/c-pairs combine in a circular way, the conclusion is not valid unless further knowledge is included in the proof. Consider an example system $S = S_1||S_2$ for which the following holds:

$$\begin{aligned} S_1 &\models (a_1, c_1), \quad c_1 \Rightarrow a_2, \\ S_2 &\models (a_2, c_2), \quad c_2 \Rightarrow a_1. \end{aligned} \tag{6}$$

Since for logical expressions a and b

$$(a \Rightarrow b) \wedge (b \Rightarrow a) \equiv (a \wedge b) \vee (\neg a \wedge \neg b), \tag{7}$$

it can only be deduced from (6) that S_1 and S_2 either both fulfill their commitments or don't: $S_1 || S_2 \models \big(true, (c_1 \wedge c_2) \vee (\neg c_1 \wedge \neg c_2)\big)$.

If circularity occurs, it must be broken by including appropriate additional conditions B. *Temporal induction* can be used to solve this problem (2): First, it is shown that in its initial state $S_1 || \ldots || S_n \models a_1, \ldots, a_n$. In the induction step it must be established that given valid commitments c_i no transition occurring in the system can violate any of the a_{i+1}. This relates to (7) as:

$$\big(a_0 \wedge b_0 \wedge \forall k \in \mathbb{N}.(a_k \Rightarrow b_{k+1}) \wedge (b_k \Rightarrow a_{k+1})\big) \Rightarrow \big(\forall k \in \mathbb{N}.a_k \wedge b_k\big) \tag{8}$$

In summary, the aim of the assumption/commitment-paradigm is to combine a/c-pairs (a_i, c_i) with additional conditions B to the following proof rule:

$$\frac{\begin{array}{c} S_1 \models (a_1, c_1) \\ \vdots \\ S_n \models (a_n, c_n) \\ B(a_1, \ldots, a_n, c_1, \ldots, c_n, a, c) \end{array}}{S_1 || S_2 || \ldots || S_n \models (a, c)}. \tag{9}$$

The selection of appropriate a/c-pairs is the creative task of the analyst and difficult to automate. The following section describes how to automate the verification of the individual a/c-pairs. Afterwards, two paradigms are presented that can provide a starting set of a/c-pairs that can then be modified to suit the particular application (20).

Application using automata The a/c-pairs (4) can be verified automatically if they are represented by automata. Let A_i be the automaton that represents the behaviors of the environment E_i fulfilling a_i and C_i be the automaton that represents all behaviors of S_i that fulfill c_i:

$$\begin{aligned} &A_i \preceq E_i \wedge A_i \models a_i, \\ &C_i \preceq S_i \wedge C_i \models c_i. \end{aligned} \tag{10}$$

A_i and C_i can be obtained manually from E_i and S_i, or by using the tableau construction $A_i = T_{a_i}, C_i = T_{c_i}$. Then (4) is equivalent to

$$A_i || S_i \preceq A_i || C_i. \tag{11}$$

This inequality can be verified with a model checking tool using a test automaton construction (3).

Chain Proof Rule In a chain rule form, which was used in the beginnings of a/c reasoning (50), the assumption/commitment proof becomes simple and requires no further additional logical conditions or explicit deduction:

$$\frac{\begin{array}{c} S_1 \preceq \hat{S}_1 \\ \hat{S}_1||S_2 \preceq \hat{S}_1||\hat{S}_2 \\ \vdots \\ \hat{S}_1||\hat{S}_2||\ldots||\hat{S}_{n-1}||S_n \preceq \hat{S}_1||\hat{S}_2||\ldots||\hat{S}_n \end{array}}{S_1||S_2||\ldots||S_{n-1}||S_n \preceq \hat{S}_1||\hat{S}_2||\ldots||\hat{S}_n}. \tag{12}$$

It can be interpreted in the following way: $\hat{S}_1$ has to capture the behavior of S_1 for all possible inputs. $\hat{S}_2$ has to simulate S_2 with the inputs from $\hat{S}_1$, which is easier than with all possible inputs. For the last module $\hat{S}_n$, only the behavior occurring under the influence of $\hat{S}_1||\ldots||\hat{S}_{n-1}$ has to be taken into account.

The proof of (12) is straightforward and can be done by iteratively applying the equations to their successors. This rule is simple, but in the following sense, it can't be improved:

- Adding a term $\hat{S}_i$ to both sides of one of the equations will destroy the soundness unless further conditions are included.
- Removing a term $\hat{S}_{i+1}$ will lead to a wider range of inputs that S_i will have to cooperate with.

Let A denote an automaton modeling a global assumption as part of the initial conditions. The automata A_i and C_i become:

$$\begin{aligned} &A_1 = A, \quad A_i = \hat{S}_1||\ldots||\hat{S}_{i-1} \text{ for } i > 1, \\ &C_i \preceq \hat{S}_i. \end{aligned} \tag{13}$$

In order to reduce the complexity of the proof steps, the assumption can be widened, i.e., for $j < i$ any $\hat{S}_j$ can be dropped from both sides of (13) at any step. This however might lead to an abstraction that is too wide and violates one of the proof steps. If the proof fails because the interactions of the modules cannot be captured by the abstractions in a chain sequence, the assumption should be made more restrictive by adding any S_j, $j > i$, to both sides of (13) at any step. This in turn will increase the complexity.

Circular Proof Rule The following proof rule, also referred to as Assume/Guarantee rule, has successfully been applied to small real-time and hybrid systems (24).

In order to verify that $S_i||\dots||S_n$ meets the specifications $\hat{S}_i||\dots||\hat{S}_n$ the following proof is carried out:

$$\frac{\begin{array}{c} S_1||\hat{S}_2||\dots||\hat{S}_{n-1}||\hat{S}_n \preceq \hat{S}_1||\hat{S}_2||\dots||\hat{S}_n \\ \hat{S}_1||S_2||\dots||\hat{S}_{n-1}||\hat{S}_n \preceq \hat{S}_1||\hat{S}_2||\dots||\hat{S}_n \\ \vdots \\ \hat{S}_1||\hat{S}_2||\dots||\hat{S}_{n-1}||S_n \preceq \hat{S}_1||\hat{S}_2||\dots||\hat{S}_n \\ B(S_1,\dots,S_n,\hat{S}_1,\dots,\hat{S}_n) \end{array}}{S_1||S_2||\dots||S_{n-1}||S_n \preceq \hat{S}_1||\hat{S}_2||\dots||\hat{S}_n}. \tag{14}$$

Additional conditions B are needed to avoid that the composition of the original modules shows a behavior that can't be met by more than one of the abstractions, in which case the proof would fail. Temporal induction can be applied to accomplish soundness of the proof (2).

With the following definition for A_i and C_i, the constituents of (14) can be obtained from (11):

$$\begin{aligned} &A_i = \hat{S}_1||\dots||\hat{S}_{i-1}||\hat{S}_{i+1}||\dots||\hat{S}_n, \\ &C_i \preceq \hat{S}_i. \end{aligned} \tag{15}$$

4.4 Example

The following example shall illustrate the above methodology. The delivery of raw materials (educts) for a chemical batch process must be in tune with the downstream reactor schedule. In a decentralized control scheme, the delivery schedule can be set within certain limits that guarantee compatibility with the downstream recipe. Once those limits are set, the downstream must in turn consume the delivered raw materials in time.

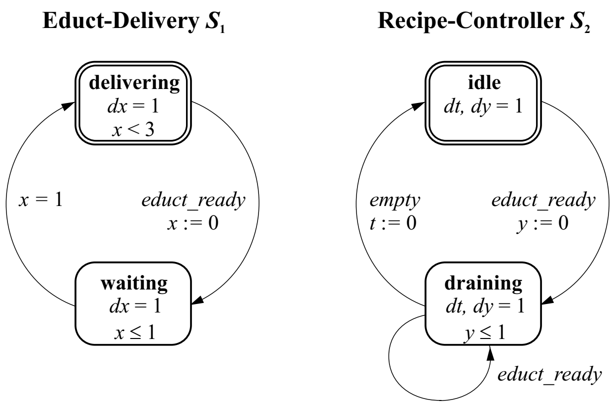

Fig. 4. Automata for the delivery and the recipe controller

Such a delivery schedule S_1 can be modeled by a timed automaton as shown in Fig. 4. The delivery takes place at least every 3 min., which in the automaton is represented by the invariant $x < 3$ in the initial state, indicated by the double line. When the educt is stored in a buffer tank, the delivery schedule provides a signal, represented by the label *educt_ready*, to the controllers and remains in a waiting state in order to give the recipe controller time to drain the buffer tank. The guard $x = 1$ on the transition back to the initial state forces the automaton to wait exactly 1 min. before the next delivery can take place.

A second automaton S_2 in Fig. 4 models the recipe controller. It remains in an *idle* state until the *educt_ready* signal is issued. The controller then goes into the state *draining* in which the valves are open to drain the educt into the reactors. To avoid including a tank model, the draining process has been abstracted by allowing the controller to stay in the draining state at most 1 min., which is assumed to be the maximum time needed for draining. This is implemented using the clock y that is reset at the transition labeled with *educt_ready*. The invariant $y \leq 1$ forces the automaton to leave the state *draining* after 1 min. at the latest and proceed to send the *empty* signal and return to the initial state. A clock t has been included to measure the time that elapses between two consecutive *empty* signals.

The requirement to be verified is whether the buffer tank is emptied at least once every 4 min. In temporal logic, this can be represented as a commitment:

$$c_2 = \forall\Box\,(t < 4) \tag{16}$$

The automaton representation C_2 corresponding to c_2 is shown in Fig. 5 together with the test automaton C_2^T that is used to verify $S_1||S_2 \preceq C_2$ using a model checking tool.

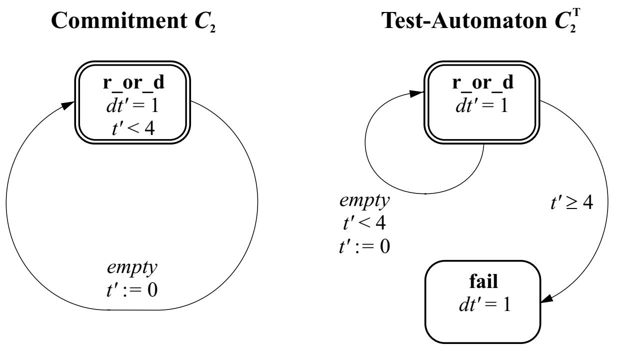

Fig. 5. Automata for representing and for testing the commitment of the recipe controller

In this example, a chain rule deduction is carried out. Starting with the requirement (16) it must be shown that

$$S_1||S_2 \models (true, c_2), \tag{17}$$

From a local analysis of S_2 the assumption a_2 as represented in Fig. 6 was obtained and it was verified that indeed $S_2 \models (a_2, c_2)$ using model checking by showing that $A_2||S_2||C_2^T \models \neg reach(fail)$. Finally it must be shown that S_1 fulfills the assumption

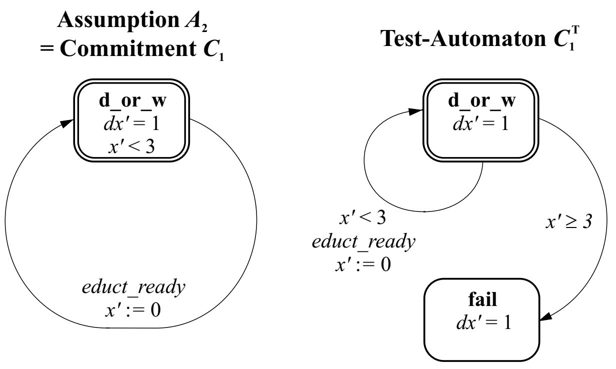

Fig. 6. Automata for representing and for testing the commitment of the delivery schedule

a_2. To do so, the commitment of S_1 is set equal to a_2 and it remains to prove $S_1 \models (true, c_1)$. This is checked using the test automaton C_1^T from Fig. 6 to show that $S_1||C_1^T \models \neg reach(fail)$.

Finally, it can be deduced in either a/c or abstraction representation that the requirement c_2 holds also for the composed system:

$$\frac{\begin{array}{c} S_1 \models (true, c_1) \\ S_2 \models (a_2, c_2) \\ c_1 = a_2 \end{array}}{S_1||S_2 \models (true, c_2)}, \qquad \frac{\begin{array}{c} S_1 \preceq C_1 \\ A_2||S_2 \preceq A_2||C_2 \\ C_1 \preceq A_2 \end{array}}{S_1||S_2 \preceq C_1||C_2}. \tag{18}$$

4.5 Related Work

Since it is obvious that noncompositional methods will always be limited by the state explosion problem, there is currently very active research in the field of compositional verification. Tom Henzinger et al. use an assume-guarantee principle in hierarchical hybrid system design which supports nesting of parallel and serial composition (25). Rajeev Alur et al. use the modeling language CHARON for the modular design of interacting hybrid systems addressing different aspects of hierarchy (1). Ranjit Jhala and Ken McMillan employ compositional model checking for the verification of a processor microarchitecture (31).

Henrik Ejersbo Jensen et al. have worked on compositionality and abstraction in the field of timed automata and presented several case studies using the UPPAAL tool (29, 30).

5 Conclusions

Formal verification completes the classical tools of simulation and analysis of controlled processes by providing a conservative and correct way to analyze the functioning of the system. The drawback of the existing methods lies in their consumption of either manual work or computational resources. Compositional verification methods reduce the costly step of model checking to the composition of relatively simple subsystems. This article summarized some of the compositional approaches to verification in literature and illustrated the methods with an example. Present approaches for model checking rely on the formalisms of timed or linear hybrid automata. Techniques to approximate nonlinear hybrid systems by linear hybrid automata are available, see e.g. (55). An application to a chemical process has shown promising results (21).

Acknowledgements

This investigation was funded by the German Research Council (DFG) through the grant EN152/29-1 and RO1122/7-1. The authors wish to thank the writers of the proposal, Yassine Lakhnech and Stefan Kowalewski.

References

1. R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivanĉić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. In T.A. Henzinger and C.M. Kirsch, editors, *EMSOFT 2001: First International Workshop on Embedded Software, Tahoe City, CA, USA, October 8–10, 2001*, volume 2211 of *Lecture Notes in Computer Science*, pages 14–31. Springer-Verlag, 2001.
2. Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.
3. Krzysztof R. Apt, Nissim Francez, and Willem-Paul de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980.
4. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.
5. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109–137, 1984.
6. Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for CTL*. In *LICS '95: 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26–29, 1995*, pages 388–397. IEEE Computer Society Press, 1995.
7. Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.
8. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Communications of the ACM*, 31(3):560–599, 1984.
9. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

10. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992. Preprint version published as CMU Technical Report CMU-CS-92-160.
11. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs Workshop, IBM Watson Research Center, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1982.
12. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
13. Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.
14. Willem-Paul de Roever. The need for compositional proof systems: A survey. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, Proceedings of the International Symposium COMPOS '97, Malente, Germany, September 7–12, 1997*, volume 1536 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 1998.
15. Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, November 2001.
16. Edsger W. Dijkstra. On understanding programs (EWD 264). Published in an extended version as (17), August 1969.
17. Edsger W. Dijkstra. Structured programming. In J.N. Buxton and B. Randell, editors, *Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee*, pages 84–88. NATO Science Committee, 1969.
18. David Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 12–14, 1989*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1990.
19. Robert W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceedings AMS Symposium Applied Mathematics*, volume 19, pages 19–31, Providence, RI, 1967. American Mathematical Society.
20. Goran Frehse, Olaf Stursberg, Sebastian Engell, Ralf Huuck, and Ben Lukoschus. Modular analysis of discrete controllers for distributed hybrid systems. In *b '02: The XV. IFAC World Congress, Barcelona, Spain, July 21–26, 2002*, 2002. To appear.
21. Goran F. Frehse, Olaf Stursberg, Sebastian Engell, Ralf Huuck, and Ben Lukoschus. Verification of hybrid controlled processing systems based on decomposition and deduction. In *ISIC 2001: 16th IEEE International Symposium on Intelligent Control, Mexico City, Mexico, September 5–7, 2001*, pages 150–155. IEEE Control Systems Society, IEEE Press, 2001.
22. Herman H. Goldstein and John von Neumann. Planning and coding problems of an electronic computing instrument. In A.H. Taub, editor, *J. von Neumann—Collected Works*, pages 80–151. McMillan, New York, 1947.
23. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1:110–122, 1997.
24. T.A. Henzinger, S. Qadeer, S.K. Rajamani, and S. Tasiran. You assume, we guarantee: Methodology and case studies. In *CAV '98: 10th International Conference on Computer-*

Aided Verification, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer-Verlag, 1998.

25. Thomas A. Henzinger, Marius Minea, and Vinayak Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In *HSCC '01: 4th International Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 275–290. Springer-Verlag, 2001.
26. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.
27. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Engelwood Cliffs, 1985.
28. Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
29. Henrik Ejersbo Jensen. *Abstraction-Based Verification of Distributed Systems*. PhD thesis, Aalborg University, June 1999.
30. Henrik Ejersbo Jensen, Kim Guldstrand Larsen, and Arne Skou. Scaling up Uppaal – automatic verification of real-time systems using compositionality and abstraction. In Mathai Joseph, editor, *FTRTFT 2000: 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, September 20–22, 2000, Pune, India*, volume 1926 of *Lecture Notes in Computer Science*, pages 19–30. Springer-Verlag, 2000.
31. Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In G. Berry, H. Comon, and A. Finkel, editors, *CAV 2001: 13th International Conference on Computer Aided Verification, Paris, France, July 18–22, 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 396–410. Springer-Verlag, 2001.
32. Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University Computing Laboratory, June 1981. Printed as: Programming Research Group, Technical Monograph 25.
33. Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
34. Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.
35. Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.
36. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
37. Gary M. Levin and David Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
38. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Twelfth ACM Symposium on the Priciples of Programming Languages*, pages 97– 105, 1985.
39. Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, May 1992. CMU Technical Report CMU-CS-92-131.
40. Kenneth L. McMillan. *The SMV system*. Carnegie Mellon University, November 2000. Manual for SMV version 2.5.4.
41. Stephan Merz. Model checking: A tutorial overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, 2001.
42. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

43. Robin Milner. *Communication and Concurrency*. Prentice-Hall International, Engelwood Cliffs, 1989.
44. Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.
45. Peter Naur. Proof of algorithms by general snapshots. *BIT (Nordisk tidskrift for informationsbehandling)*, 6(4):310–316, 1966.
46. A. Olivero and S. Yovine. *KRONOS: A Tool for Verifying Real-Time Systems. User's Guide and Reference Manual*. Verimag, Grenoble, France, 1993.
47. Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
48. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.
49. Amir Pnueli. The temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
50. Amir Pnueli. In transition for global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI-F*. Springer-Verlag, 1984.
51. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the 5th International Symposium on Programming, Turin, April 6–8, 1982*, pages 337–350. Springer-Verlag, 1982.
52. Wolfgang Reisig. *Petri Nets, An Introduction*. EATCS, Monographs on Theoretical Computer Science. Springer Verlag, Berlin, 1985.
53. Specification and Description Language SDL, blue book. CCITT Recommendation Z.100, 1992.
54. Karsten Stahl. Comparing the expressiveness of different real-time models. Master's thesis, Christian-Albrechts-University of Kiel, May 1998.
55. Olaf Stursberg. Analysis of switched continuous systems based on discrete approximation. In *ADPM 2000: 4th International Conference on Automation of Mixed Processes*, pages 73–78, 2000.
56. Alan M. Turing. On checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, 1949. University Mathematics Laboratory.
57. Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.