Practical Efficient Modular Linear-Time Model-Checking

Carlo A. Furia[1] and Paola Spoletini[2]

[1] DEI, Politecnico di Milano, Milano, Italy
[2] DSCPI, Università degli Studi dell'Insubria, Como, Italy

Abstract. This paper shows how the modular structure of composite systems can guide the state-space exploration in explicit-state linear-time model-checking and make it more efficient in practice. Given a composite system where every module has input and output variables — and variables of different modules can be connected — a total ordering according to which variables are generated is determined, through heuristics based on graph-theoretical analysis of the modular structure. The technique is shown to outperform standard exploration techniques (that do not take the modular structure information into account) by several orders of magnitude in experiments with Spin models of MTL formulas.

1 Introduction

Systems are complex; as apparent as it sounds, *complexity* is the primal hurdle when it comes to describing and understanding them. Abstraction and *modularization* are widely-known powerful conceptual tools to tame this complexity. In extreme summary, a large system is described as the composition of simpler modules. Every module encapsulates a portion of the system; its internal behavior is abstracted away at its interface — the set of input/output variables that are connected to other modules [14]. Modularization is widely practiced in all of computer science and software engineering.

A class of systems that are especially difficult to analyze is given by *concurrent* systems. In such systems the various parts are often highly coupled, as a result of their ongoing complex synchronization mechanisms. Nonetheless, over the last decades the state of the art in specifying and verifying concurrent systems has made very conspicuous advancements. A significant part of them is centered around the formalisms of temporal logics [4] and finite-state automata [18], and the algorithmic verification technique of model-checking [1].

Although model-checking techniques target primarily closed monolithic systems, modularization has been considered for model-checking in the research trends that go by the names *module checking* [12] and *modular model-checking* [11]. Both extend model-checking techniques to *open* systems, i.e., systems with an explicit interaction with an external environment (that provides input) [8]. Then, in module checking properties of the system are checked with respect to *all* possible environments, whereas in modular model-checking properties are checked with respect to environments satisfying a temporal-logic specification (according to the *assume/guarantee* paradigm [3]).

In this paper we take a different approach, which exploits the information that comes from the modular decomposition of systems to ameliorate model-checking performances in practice. We consider explicit-state model-checking techniques for linear-time temporal logic: the system and the property are represented as finite-state automata, and checking that all runs of the system satisfy the property amounts to performing an exploration of the state-space of the overall automaton — resulting from the composition of the various component automata — in order to detect cycles (which correspond to runs where the property is violated) [1]. This exploration is the more efficient the earlier we are able to detect "unproductive" paths that lead to no cycle. If the various components of a system are decomposed into communicating modules, the information about how these modules are connected is useful to guide such state-space exploration paths.

Our approach aims at being practical, in that we do not claim any asymptotic worst-case gain over traditional algorithms. In fact, our technique is essentially based on heuristics that may or may not be effective according to the particular structure of the system at hand, and that cannot escape the inherent worst-case complexity of automated verification. However, we demonstrate the significant practical impact of our technique by means of a verification examples where traditional "vanilla" techniques are compared against our optimized modular approach. Our technique clearly outperforms the unoptimized algorithm by several orders of magnitude.

The rest of the paper is organized as follows. In the next sub-section we briefly review some related work. Section 2 introduces most definitions that will be used in the rest of the paper, including our notion of module. Section 3 describes the optimization technique itself in detail, after an informal overview. Section 4 introduces the verification examples and reports on experimental results with them. Finally, Section 5 concludes and hints at future work.

Related work. In the Introduction we already mentioned the techniques of module checking and modular model-checking. For linear-time temporal logic, module checking and modular model-checking basically reduce to standard model-checking, as discussed in [12] and [11], respectively. In fact, any linear-time environment can simply be described as an explicit component of the system, thus reducing to the usual case of closed monolithic systems. This entails that the complexity of the module-checking and modular model-checking problems is not different than ordinary linear-time model-checking; in fact, they all are PSPACE-complete problems.

In practice, however, these modular techniques have proved to be extremely effective in taming the state-explosion problem which plagues model-checking. Compositionality and the *assume/guarantee* paradigm, in particular, can be seen as an application of the abstraction and modularization principles to the specification and verification of concurrent systems: the module is decoupled from its environment by abstracting the latter as an assumption formula. Verification can then be performed locally to the module, without resorting to the full description of its environment. There is quite a large amount of literature dealing with the subject of compositionality; we refer the reader to [3, 5] for surveys and further references.

The work in this paper is most directly based upon [16], where topological information that comes from the modular decomposition is used to drive the systematic generation of test-case execution sequences for real-time systems described in metric temporal logic. Previous work of ours [13, 17] explored the possibility of using similar techniques for enhancing practical performances of linear-time model-checking. In Section 3 we develop systematically those preliminary ideas into a comprehensive technique. In particular, the examples in Section 4 are based on previous work on the translation of metric temporal logic (such as MTL and TRIO) models into ProMeLa models, which can be analyzed with Spin [17, 13, 15].

2 Definitions

2.1 Variables and Computations

A *variable* v is characterized by the finite domain D_v over which it ranges; if no domain is specified the variable is assumed to be Boolean with $D_v = \{0, 1\}$. For a set of variables $\mathcal{V}$, $\mathcal{V}'$ denotes the set of primed variables $\{v' \mid v \in \mathcal{V}\}$ with the same domains as the original variables.

The behavior of systems — and components thereof — is described by ω-sequences of variable values called computations. Formally, given a finite set of variables $\mathcal{V}$, a *computation* over $\mathcal{V}$ is an ω-sequence $w = w_0, w_1, w_2, \ldots \in D^\omega$, where D is the Cartesian product $\prod_{v \in \mathcal{V}} D_v$ of variable domains. Also, given a subset of variables $Q \subseteq \mathcal{V}$, the *projection* of w over Q is a computation $x = x_0, x_1, x_2, \ldots$ over Q obtained from w by dropping the components of variables in $\mathcal{V} \setminus Q$, that is $x_j = w_j|_Q$ for all $j \geq 0$. Projection is extended to sets of computations as obvious: for a set of computations C, its projection over Q is $C|_Q = \{w|_Q \mid w \in C\}$. The set of all computations over $\mathcal{V}$ is denoted by $\mathcal{C}(\mathcal{V})$.

2.2 LTL and MTL

In the examples of Section 4 we are going to express the behavior of modules by means of Metric Temporal Logic (MTL) formulas. This section introduces the syntax and semantics of MTL.

Let $\mathcal{V}$ be a set of variables. MTL formulas are given by: $\phi ::= v = \mathrm{c} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{U}_J(\phi_1, \phi_2) \mid \mathsf{S}_J(\phi_1, \phi_2)$, for $v \in \mathcal{V}$, $\mathrm{c} \in D_v$, and $J \subseteq \mathbb{N}$ an interval of the nonnegative integers. The basic temporal operator is the *bounded until* U_J (and its past counterpart *bounded since* S_J); it is the metric version of the well-known LTL until operator.

MTL formulas are interpreted over computations over the set $\mathcal{V}$ of variables. Given such a computation w, a time instant $t \geq 0$, and an MTL formula ϕ, the satisfaction relation $\models$ is defined as follows:[3]

[3] We assume that $w, t^- \not\models \phi$ for all $t^- < 0$.

$$
\begin{array}{lll}
w,t \models v = \mathrm{c} & \text{iff} & w(t)|_v = \mathrm{c} \\
w,t \models \neg\phi & \text{iff} & w,t \not\models \phi \\
w,t \models \phi_1 \wedge \phi_2 & \text{iff} & w,t \models \phi_1 \text{ and } w,t \models \phi_2 \\
w,t \models \mathsf{U}_J(\phi_1,\phi_2) & \text{iff} & \exists d \in J : (w,t+d \models \phi_2 \wedge \forall 0 < u < d : w,t+u \models \phi_1) \\
w,t \models \mathsf{S}_J(\phi_1,\phi_2) & \text{iff} & \exists d \in J : (w,t-d \models \phi_2 \wedge \forall 0 < u < d : w,t-u \models \phi_1) \\
w \models \phi & \text{iff} & w,0 \models \phi
\end{array}
$$

Whenever $w \models \phi$ we say that w satisfies ϕ. Any MTL formula identifies a set of computations $L(\phi) = \{w \in \mathcal{C}(\mathcal{V}) \mid w \models \phi\}$ called the language of ϕ.

Standard abbreviations are assumed such as $\top, \bot, \vee, \Rightarrow, \Leftrightarrow$. In addition, we introduce the following derived temporal operators: the (bounded) *eventually* $\mathsf{F}_J(\phi) \triangleq \mathsf{U}_J(\top, \phi)$, the (bounded) *globally* $\mathsf{G}_J(\phi) \triangleq \neg\mathsf{F}_J(\neg\phi)$, the *next* $\mathsf{X}(\phi) \triangleq \mathsf{F}_{=1}(\phi)$, the (bounded) *previously* $\mathsf{P}_J(\phi) \triangleq \mathsf{S}_J(\top, \phi)$, the *yesterday* $\mathsf{Y}(\phi) \triangleq \mathsf{P}_{=1}(\phi)$, the *always* $\mathrm{Alw}(\phi) \triangleq \mathsf{G}_{(0,\infty)}(\phi) \wedge \phi$. Note that, whenever no interval is specified, $I = (0, \infty)$ is assumed; also, intervals are abbreviated by pseudo-arithmetic expressions such as $= k, \geq k, < k$ for $[k,k], [k,\infty), (0,k)$ respectively. It should be clear that, over computations, MTL is just LTL with syntactic salt: LTL's *next* operator X can be used to "count" distances over discrete time.

2.3 Modules and Composition

A system is described by the composition of modules; $\mathcal{M}$ denotes the set of all modules.

Primitive modules. The simplest component is the *primitive module*, defined as $M = \langle I, O, H, W \rangle$, where:

- I, O, and H are sets of input, output, and hidden (i.e., internal) variables, respectively. We assume that these sets are pairwise disjoint. $P = I \cup O \cup H$ denotes all variables of the module.
- W is a set of computations over P, describing the module's semantics. In practice, the behavior of modules is specified as the language $L(F)$ of some finite-state automaton or temporal logic formula F.
  Usually, one assumes that the value of input variables is provided "from the outside", hence it should not be constrained in W; this can be stated formally by requiring that $W|_I \triangleq \{w|_I \mid w \in W\}$ equals $\mathcal{C}(I)$. However, this assumption is not strictly required for the discussion of this paper, as it will be clear in the following.

We introduce a graphical representation for (the interface of) primitive modules: a module is represented by a box with inward arrows corresponding to variables in I, outward errors corresponding to variables in O, and internal lines corresponding to variables in H.

Example 1. Primitive module M_3, pictured in Figure 1, has input variables $I = \{\mathsf{v}_4, \mathsf{v}_8\}$, output variables $O = \{\mathsf{v}_5, \mathsf{v}_6, \mathsf{v}_7\}$, and hidden variables $H = \{\mathsf{v}_{11}, \mathsf{v}_{12}\}$.

Composite modules. Primitive modules can be composed to build *composite modules*. A composite module is defined as $N = \langle I, O, n, \eta, C, X \rangle$, where:

- $n > 0$ is the number if internal modules;

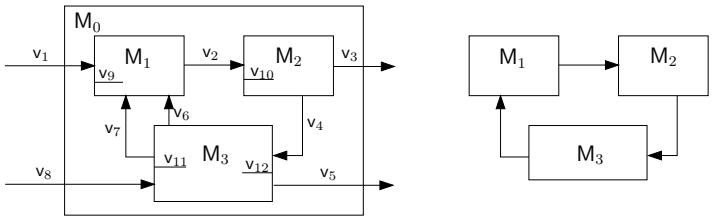

Fig. 1. Flat composite open module M_0 (left) and its connection graph (right).

- η is a finite set of module identifiers such that $|\eta| = n$;
- $C : \eta \to \mathcal{M}$ provides the module definition $C(i)$ of every internal module $i \in \eta$. We denote the components of every module $C(i)$ with superscripts as in I^i, O^i, H^i, etc. Also, we define the sets of all input, output, and hidden variables of internal modules as: $\mathcal{I} \triangleq \bigcup_{i\in\eta} I^i$, $\mathcal{O} \triangleq \bigcup_{i\in\eta} O^i$, and $\mathcal{H} \triangleq \bigcup_{i\in\eta} H^i$ respectively. Accordingly, $\mathcal{V} \triangleq \mathcal{I} \cup \mathcal{O} \cup \mathcal{H}$.
- $X \subseteq \mathcal{O} \times \mathcal{I}$ is a connection relation, which defines how the inputs and outputs of the various modules are connected: $(o, i) \in X$ iff output o is connected to input i.
- I, O have the same meaning as in primitive modules. Hence, input and output variables of composite modules are defined as those of internal modules that are not connected, namely: $I = \{i \in \mathcal{I} \mid \forall o \in \mathcal{O} : (o, i) \notin X\}$ and $O = \{o \in \mathcal{O} \mid \forall i \in \mathcal{I} : (o, i) \notin X\}$.

We extend the graphical notation to composite modules as obvious, by representing connections through connected arrows.

Modules classification. A module is *closed* iff $I = \emptyset$, otherwise it is *open*. A module is *flat* iff it is primitive or it is composite and all its internal modules are primitive; if a module is not flat it is *nesting*. A module is *non-hierarchical* iff it is flat or it is nesting and all its components are flat; otherwise it is *hierarchical*.

For a composite module $N = \langle I, O, n, \eta, C, X\rangle$, its *connection graph* is a directed graph $G = \langle V, E\rangle$ with $V = \eta$ and $(h, k) \in E$ iff there is a connection $(o, i) \in X$ with $o \in O^h$ and $i \in I^k$. We stretch the terminology by "lifting" attributes of the connection graph to the modules themselves. So, for instance, if the connection graph is acyclic (resp. connected), the modular system is called acyclic (resp. connected), etc.

Example 2. Figure 1 (left) pictures flat composite open module M_0 with $I = \{\mathsf{v}_1, \mathsf{v}_8\}$, $O = \{\mathsf{v}_3, \mathsf{v}_5\}$, $n = 3$, $\eta = \{\mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_3\}$. For graphical simplicity, variables that are connected are given a unique name. To the right, we have the connection graph of M_0.

Modules semantics. Let us define the *semantics* of modules. For a primitive module M, the semantics is trivially given by $W = L(M)$, which is called the *language* of M.

Let us now consider a composite module N. The language $L(N)$ accepted by such a module is a set of computations over $\mathcal{V}$ defined as follows. A computation w is in $L(N)$ iff: (1) w is compatible with every component module, i.e., $w|_{P^i} \in L(M^i)$ for

all component modules $i \in \eta$; and (2) connections between modules are respected, i.e., for all connections $(o, i) \in X$ we have $w|_{\{o\}} = w|_{\{i\}}$.

Notice that, for linear-time models, semantics of open modules is trivial, and implicit in our previous definitions. To make this apparent, we introduce the notion of maximal environment, which is a module generating all possible inputs to another (open) module. Given a set V of variables, a *maximal environment* $\mathcal{E}(V)$ is a primitive module such that $I = H = \emptyset$, $O = V$, and the language $L(\mathcal{E}(V))$ is exactly $\mathcal{C}(V)$. So, for an open module K (either primitive or composite), the language $L(K)$ can be defined as the language of the composite closed module K' obtained by composing K with a maximal environment. Hence, $K' = \langle \emptyset, O^K, 2, \{\mathrm{e}, \mathrm{m}\}, \langle \mathcal{E}(I^{K'}), K \rangle, X \rangle$ with $X = \{(x', x) \mid x \in I^K\}$. However, for any computation $x \in \mathcal{C}(\mathcal{V} \cup I^{K'})$ it is $x \in L(K')$ iff $x|_{I^{K'}} \in L(\mathcal{E}(I^{K'})) = \mathcal{C}(I^{K'})$ and $x|_{\mathcal{V}} \in L(K)$ and $x|_{I^{K'}} = x|_{I^K}$. Hence, $L(K')|_{\mathcal{V}} = L(K)$.

3 Efficient Design of Generators

3.1 Practical Module Checking

Let us consider what happens in practice when performing explicit-state model-checking of a modular system using an automata-based approach. In this setting, the model-checking algorithm is basically an on-the-fly state-space search for cycles (or absence thereof). Correspondingly, the modular structure of the system can be exploited to greatly improve the performances of the check in practice. Essentially, structure can guide the state-space exploration in order to minimize the degree of unnecessary non-determinism.[4] We introduce this technique with an overview example.

Example 3. Consider the example of a beverage vending machine, adapted from [12]. The machine can be modeled as a primitive module M with $I^{\mathsf{M}} = \{\mathsf{idle}, \mathsf{start}, \mathsf{choice} : \{\mathrm{tea}, \mathrm{coffee}\}\}$ and $O^{\mathsf{M}} = \{\mathsf{done} : \{\mathrm{tea}, \mathrm{coffee}, \bot\}\}$ whose behavior is defined by the finite state automaton in Figure 2. Consider a client module C with $I^{\mathsf{C}} = \{\mathsf{idle}, \mathsf{start}\}$ and $O^{\mathsf{C}} = \{\mathsf{choice} : \{\mathrm{tea}, \mathrm{coffee}\}\}$ that always chooses tea, i.e., whose behavior is defined by the LTL formula $\mathsf{start} \Rightarrow \mathsf{X}(\mathsf{choice} = \mathrm{tea})$. It is clear that the composition of the two modules (with a generic environment $\mathcal{E}$) as in Figure 2 guarantees property $\pi = \mathrm{Alw}(\mathsf{done} \neq \mathrm{coffee})$.

In principle, a state exploration process could start generating any possible choice for the various variables, one step at a time, until it realizes that the states corresponding to property π cannot be reached. Such a process would also consider computations with start followed in the next state by $\mathsf{choice} = \mathrm{coffee}$; these sequences are not compatible with the behavior of module C and thus the corresponding path must be aborted. However, if the state space exploration is random, it could happen that the semantics of C is considered last, thus the unfeasible paths are pruned after a significant amount of processing has been done to no avail. On the contrary, generating variables according

[4] In a sense, a model with shared-variable concurrency is transformed into one with message-passing concurrency, according to the functional dependencies among variables of different modules.

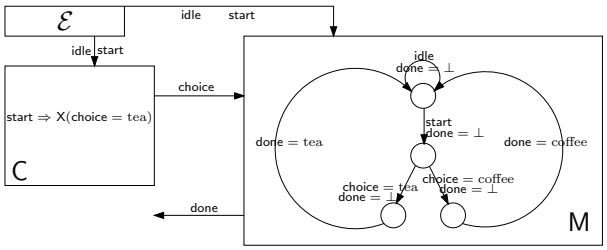

Fig. 2. Beverage machine example.

to the input/output relations among modules yields a more efficient exploration. In the example, the generation could start from the environment, pass through module C and then pass its output variables to module M. This means that computations with start followed in the next state by choice $=$ coffee would not "reach" module M, hence pruning down unfeasible paths as soon as possible.

We take advantage of these remarks in the following way. For every module M in a system we introduce a *generator* component $\mathcal{G}(M)$. The generator is responsible for setting the value of all variables in M. It operates as an interface between M and the other modules in the system. Namely, it can receive input variables from the other modules, once they have generated them, and it is responsible for setting the value of hidden and output variables, according to the behavior of M. We also define a *total ordering* over all generators in a system. This induces a generation order for environment variables in the whole system. As we have shown in the previous example, this can influence the efficiency of the system state-space exploration.

Notice that generators are not additional modules of the system, but they are components that pertain to a lower level of abstraction, namely the system description in the model-checking environment. These components realize in practice the coordination among modules in an efficient way. This framing of the problem has been especially inspired by our experience with the Spin model-checker and its implementation of ProMeLa processes [9], in particular the one based on a translation from TRIO metric temporal logic formulas [17, 13, 15]. However, we present the results in a more general setting which is exploitable also with other linear-time explicit-state model-checkers.

Example 4. Let us go back to the beverage vending machine example, in order to illustrate practically the idea of generator. Table 1 outlines a portion of ProMeLa code for the generator $\mathcal{G}(\mathcal{E})$ of the environment. The generator first of all waits for messages from the module that immediately precedes it in the global ordering (line 2); since $\mathcal{E}$ is the first in our ordering, this happens at the beginning of every full round in the state-space exploration process. When a non-error "acknowledgement" message is received the actual generation process is started (lines 10–19). For every variable, a value in its domain is chosen nondeterministically and it is assigned to the variable itself (e.g., lines 12–13 for the Boolean variable idle). After a full set of values has been generated, it is checked for consistency with the constraints induced by the module's semantics. Since $\mathcal{E}$ is a maximal environment (see definitions above) this step is absent in the example

```
1  do
2    :: msg?x,eval(generator_id);
3    if
4      :: x==0
5        /* error occurred */
6        -> goto error_handling_routine;
7      :: else
8        /* no error, go on with actual generation */
9        ->
10          if
11            /* generate values for "idle" */
12            :: idle=0;
13            :: idle=1;
14          fi;
15          if
16            /* generate values for "start" */
17            :: start=0;
18            :: start=1;
19          fi;
20          msg!1,C_module_proc_id;
21          fi;
22 od;
```

Table 1. ProMeLa code generator sample.

of Table 1. Finally (line 20), the generator terminates successfully its execution round by releasing its control to the next process. Since we want the generation to go on with module C, the "acknowledgement" message is sent precisely to C's generator.

In the remainder, we show a strategy to design an ordered set of generator for any given modular system. The strategy aims at designing and ordering the generators so as to cut down the state-space exploration as soon as possible. It is based on a set of heuristics and built upon the analysis of the modular structure of the system.

Clearly, we can assume that the the connection graph of our system is connected. In fact, if it is not connected, we can partition it into a collection of connected components, such that every connected component can be treated in isolation as discussed below.

3.2 Acyclic Flat Modules

Let $M = \langle I, O, n, \eta, C, X \rangle$ be an acyclic flat connected module; without loss of generality we assume it is a composite module (otherwise, just consider a composite module with a single primitive component). For every $i \in \eta$ the generator $\mathcal{G}(i)$ of module $C(i)$ is responsible for generating the following variables: $H^i \cup O^i \cup (I^i \cap I)$. That is, $\mathcal{G}(i)$ generates all hidden and output variables, and all input variables that are not connected to any output variables of other modules.

3.3 Cyclic Flat Modules

Let $M = \langle I, O, n, \eta, C, X \rangle$ be a cyclic flat connected module; note that such a module is also necessarily composite. In order to design generators for such a module we recall the notion of *feedback arc set*. Let $G = \langle V, E \rangle$ be the cyclic connection graph of M. A feedback arc set (FAS) is a set of edges $F \subseteq V$ such that the graph $\langle V, E \setminus F \rangle$ is acyclic.

In practice, we can consider M as an acyclic module with (self-)connections going from some of its output variables to some of its input variables; these connections correspond to edges F of the FAS. It is clear that a FAS always exists for a cyclic module; in general, however, the FAS is not unique.

Through the definition of FAS we can re-use the simple strategy for designing generators that we applied in the acyclic case. Namely, let $I^F \subseteq \mathcal{I} \setminus I$ and $O^F \subseteq \mathcal{O} \setminus O$ be the sets of input and output variables, respectively, corresponding to the edges in F. Then, for every $i \in \eta$ generator $\mathcal{G}(i)$ of module $C(i)$ is responsible for generating the following variables: $H^i \cup (O^i \cap (\mathcal{O} \setminus O^F)) \cup (I^i \cap I) \cup (I^i \cap I^F)$. That is, $\mathcal{G}(i)$ generates all hidden variables, all output variables that are not in the cycle (because these are the same as the input variables they are connected to, and these input variables are generated by the generator of the corresponding modules), all input variables that are not connected to any output variables of other modules (hence coming from the environment), and all input variables that belong to the FAS.

Example 5. Consider the connection graph of cyclic module M_0 in Figure 1. If we choose $F_1 = \{(\mathsf{M}_3, \mathsf{M}_1)\}$ as FAS, the generators would generate the following variables: $\mathcal{G}(\mathsf{M}_1) = \{\mathsf{v}_9, \mathsf{v}_2, \mathsf{v}_1, \mathsf{v}_6, \mathsf{v}_7\}$, $\mathcal{G}(\mathsf{M}_2) = \{\mathsf{v}_{10}, \mathsf{v}_3, \mathsf{v}_4\}$, $\mathcal{G}(\mathsf{M}_3) = \{\mathsf{v}_{11}, \mathsf{v}_{12}, \mathsf{v}_5, \mathsf{v}_8\}$. If we choose instead $F_2 = \{(\mathsf{M}_1, \mathsf{M}_2)\}$ as FAS, we would generate: $\mathcal{G}(\mathsf{M}_1) = \{\mathsf{v}_9, \mathsf{v}_1\}$, $\mathcal{G}(\mathsf{M}_2) = \{\mathsf{v}_{10}, \mathsf{v}_3, \mathsf{v}_4, \mathsf{v}_2\}$, $\mathcal{G}(\mathsf{M}_3) = \{\mathsf{v}_{11}, \mathsf{v}_{12}, \mathsf{v}_5, \mathsf{v}_6, \mathsf{v}_7, \mathsf{v}_8\}$.

In order for the generation to be correct all variables in the system must be generated, in some order, in such a way that all constraints imposed by the modules' semantics are satisfied. Any FAS guarantees a correct generation in this sense, because it simply induces a particular generation order on the set of all variables, such that no variable is ignored. While correctness is guaranteed regardless of which FAS is chosen, it is advisable to choose the arcs corresponding to the minimum number of variables, so that the minimum number of variables is generated first. Hence, we introduce the following minimization problem to select a suitable FAS.

Consider the *weighted connection graph*, a weighted enhancement of the connection graph defined as follows. Let $G = \langle V, E \rangle$ be the (unweighted) connection graph. The corresponding weighted version $G_{\mathcal{W}} = \langle V, E, \mathcal{W} \rangle$ introduces a weight function $\mathcal{W} : E \to \mathbb{N}_{>0}$ that associates with every edge $e = (M_1, M_2) \in E$ a weight $\mathcal{W}(e) = \prod_{v \in M_1 \succ M_2} |D_v|$ where $M_1 \succ M_2$ is the set of output variables of M_1 connected to input variables of M_2 (i.e., $M_1 \succ M_2 = \{o \in O^{M_1} \mid \exists i \in I^{M_2} : (o, i) \in X\}$).

Finding the optimal generator design amounts to solving the (weighted) minimum FAS problem over the weighted connection graph. This problem is well-known to be NP-complete [10, 6], while it is solvable in polynomial time for planar graphs [7]. However, the connection graph of a modular system is not likely to be significantly large, hence it is acceptable to use exact algorithms that have a worst-case exponential running time. Indeed, one can solve the problem with a brute-force algorithm which finds the minimum FAS $\textsc{minFAS}(G)$ for a weighted connection graph $G = \langle V, E, \mathcal{W} \rangle$ in time $\mathrm{O}(2^{|E|}|V|^2)$.

Example 6. For module M_0 in Figure 1, the weighted minimum FAS problem suggests to choose $(\mathsf{M}_1, \mathsf{M}_2)$ (or $(\mathsf{M}_2, \mathsf{M}_3)$) over $(\mathsf{M}_3, \mathsf{M}_1)$ as FAS. Notice that, if arc (m, n) is chosen, one must start generating from module n, where the broken cycle is entered.

In fact, in the previous example we have shown that choosing $(\mathsf{M}_3, \mathsf{M}_1)$ involves generating variables for modules 5, then 3, then 4, whereas choosing $(\mathsf{M}_1, \mathsf{M}_2)$ involves generating variables for modules 2, then 4, then 6.

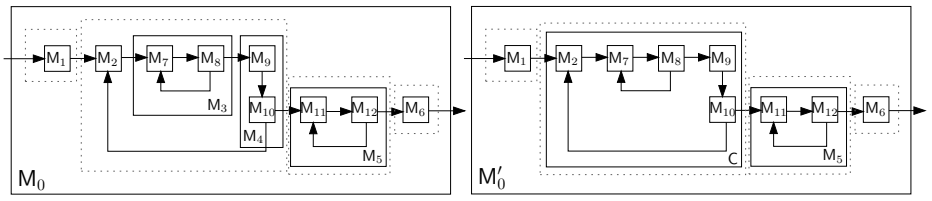

Fig. 3. Non-hierarchical nesting module M_0 and its 4 SCCs (dotted boxes).

3.4 Non-Hierarchical Nesting Modules

In increasing order of complexity, let us now consider nesting modules that are non-hierarchical. The connection graph of such modules must be first analyzed at the top level, in order to cluster its component flat modules into two classes. To this end, we have to identify the strongly connected components of the connection graph.

A *strongly connected component* (SCC) of a directed graph is a maximal sub-graph such that for every pair v_1, v_2 of its vertices there is a directed path from v_1 to v_2. The collection of all strongly connected components of a directed graph form a partition such that the "higher-level" graph where each SCC is represented by a single node is acyclic. The collection of SCCs of a graph $G = \langle V, E \rangle$ can be computed in time $\Theta(|V| + |E|)$ [2, Sec. 22.5].

For a non-hierarchical nesting module $M = \langle I, O, n, \eta, C, X \rangle$ let $G = \langle V, E \rangle$ be its connection graph, and let $S = \{S_1, S_2, \ldots, S_{|S|}\}$ be a partition of V such that $\langle S_i, E_i \rangle$ with $E_i = \{(v_1, v_2) \in E \mid v_1, v_2 \in S_i\}$ is a SCC for all $1 \leq i \leq |S|$. Then, every SCC $\langle S_i, E_i \rangle$ belongs to exactly one of the following two categories: (1) $|S_i| = 1$, that is the SCC S_i represents a single flat module; and (2) $|S_i| > 1$, that is the SCC S_i represents a collection of (more than one) flat modules. We build the generators for every module in a SCC according to the following strategy:

1. If $|S_i| = 1$ we just apply the techniques for flat modules that we presented in the previous sections;
2. If $|S_i| > 1$ we "flatten" the collection of corresponding modules as follows.
   Let $S_i \subseteq \eta$ with $|S_i| > 1$, such that every $j \in S_i$ is a flat module. Let $C = \langle I^C, O^C, n^C, \eta^C, C^C, X^C \rangle$ be a new composite module defined as follows. For every composite module $C(j) = \langle I^j, O^j, n^j, C^j, X^j \rangle$ with $j \in S_i$, we introduce in C the set of primitive modules $\{C^j(k) \mid k \in \eta^j\}$ by adding: (1) $n^j = |\eta^j|$ to n^C, (2) η^j to η^C, (3) the mappings $\{k \mapsto C^j(k) \mid k \in \eta^j\}$ to C^C, and (4) the tuples $\{(o, i) \in X^j\}$ to X^C. Also, for every primitive $C(j)$ with $j \in S_i$ we simply increase n^j by one, add the new identifier j to η^C and the new mapping

$\{j \mapsto C(j)\}$ to C^C. I^C and O^C are defined accordingly as $\bigcup_{j \in S_i} I^j$ and $\bigcup_{j \in S_i} O^j$ respectively. Finally, C replaces all modules $\{C(j) \mid j \in S_i\}$ in the system.
In all, informally, we have removed the "wrapper" of every composite module in S_i by merging its components directly into the top level of C. Now, C is a flat (composite) module, which can be analyzed through the techniques presented in the previous sections.

Example 7. Consider non-hierarchical nesting module M_0 in Figure 3 (left). Its components $\mathsf{M}_2, \mathsf{M}_3, \mathsf{M}_4$ form a SCC with more than one node, which can be flattened into module C in M_0' (right). The SCC of M_0' are the singletons $\{\mathsf{M}_1\}, \{\mathsf{C}\}, \{\mathsf{M}_5\}$, hence they can be analyzed according to the discussions in the previous section.

3.5 Hierarchical Modules

For a hierarchical module M we can apply recursively the strategies discussed in the previous sections. First, we build the connection graph for M, which represents the structure of the system at the top level. By analyzing this graph as shown before, we identify, for every node in the graph, a set of variables that must be generated for its lower-level components. Then, we recur on every node in the graph: we consider the corresponding modules in isolation from the rest, we build the corresponding (lower-level) connection graph, and further partition the variables according to the discussed techniques. In the end, we will have introduced a generator for every component at the lowest level.

3.6 Choosing the Total Ordering of Generators

Let us now discuss how to choose a total ordering over the generators. Consider a directed acyclic weighted connection graph $G = \langle V, E, \mathcal{W} \rangle$ such that for every module $M \in V$ we have defined a generator $\mathcal{G}(M)$. This setting is without loss of generality, because if the graph is cyclic we choose a FAS F as described in Section 3.3 and consider the "cut" acyclic graph $\langle V, E \setminus F, \mathcal{W} \rangle$. Also, for composite modules M we may have one generator for every component of M; however, we first consider M as an aggregate component (so $\mathcal{G}(M)$ represents a collection of generators that we consider atomic) and then recursively apply the enumeration technique to M itself.

The acyclic graph G defines the partial order $E \subseteq V \times V$ on its nodes V. Through a standard technique, we select a total order $E \subseteq \preceq \subseteq V \times V$ by repeatedly selecting a pair $M_1, M_2 \in V$ of nodes such that M_1 and M_2 are not comparable in E and adding either (M_1, M_2) or (M_2, M_1) to $\preceq$. Pairs are selected according to the *generation domain dimension* (GDD) heuristic. For a module M_i we define:[5]

$$\mathrm{gdd}(M_i) = \begin{cases} \prod_{\pi \in \Gamma^+} \mathcal{W}(\pi) + \prod_{\pi \in I^i \cap I} |D_\pi| & \text{if } M_i \text{ is a source node} \\ \prod_{\pi \in \Gamma^+} \mathcal{W}(\pi) - \prod_{\pi \in \Gamma^-} \mathcal{W}(\pi) & \text{otherwise} \end{cases}$$

with $\Gamma^+ = \{(M_i, v) \in E\}$ and $\Gamma^- = \{(v, M_i) \in E\}$ the sets of outgoing and ingoing edges, respectively, and $I^i \cap I$ the set of input variables of M_i that are generated.

[5] A source node is a node without ingoing edges.

Then, we let $M_1 \preceq M_2$ iff $\mathrm{gdd}(M_1) < \mathrm{gdd}(M_2)$. This corresponds to putting first the generators corresponding to modules that "filter out" the most variables. Hence it hopefully cuts down as soon as possible several possible future states to be considered in the state-space exploration.

4 Examples and Experiments

We introduce a modular system whose behavior is formalized by means of discrete-time MTL formulas.

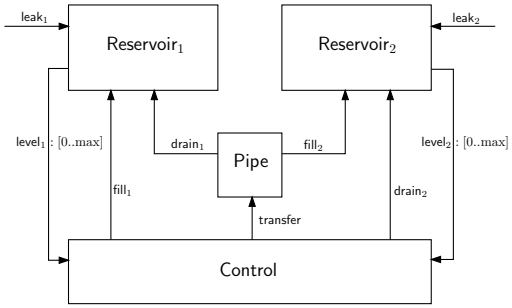

Fig. 4. The Reservoir System.

The reservoir system description. The reservoir system is made of four primitive modules: two reservoirs, a controller, and a pipe connecting the two reservoirs (see Figure 4, where the top "wrapper" module is not pictured for simplicity).

Every reservoir $\mathsf{Reservoir}_i$ ($i = 1, 2$) stores some liquid, whose level is represented at any time by variable $\mathsf{level}_i : [0..\mathrm{M}]$ where M is a constant parameter. The value of level_i changes according to the behavior of the three Boolean variables drain_i, leak_i, and fill_i, representing fluid being drained out, leaking out of, and being added to the reservoir, respectively. Correspondingly, we introduce a filling rate $\mathrm{fr}_i > 0$, a draining rate $\mathrm{dr}_i < 0$, and a leaking rate $\mathrm{dr}_i < 0$, expressed in fluid units per time unit. Leaking is completely nondeterministic, so it can happen at any time, while the other two actions are taken by the Control unit. The behavior of level is formalized by formulas (1–2) in Table 2.

The Pipe is responsible for transferring fluid from $\mathsf{Reservoir}_1$ to $\mathsf{Reservoir}_2$ upon receiving command trans; it does so simply by draining fluid from $\mathsf{Reservoir}_1$ and correspondingly filling $\mathsf{Reservoir}_2$. In order for this to make sense, we implicitly assume that $\mathrm{dr}_1 = -\mathrm{fr}_2$. The behavior of Pipe is formalized by formula (3) in Table 2.

The Control unit monitors the levels of the two reservoirs and takes actions in order to maintain both levels in the range $[1, \mathrm{M}-1]$. More precisely, the controller can perform a fill_1 on the first Reservoir, a drain_2 on the second Reservoir, or it can transfer fluid

from the first to the second Reservoir through the Pipe. The control policy can be described as follows: fill_1 is performed whenever level_1 goes below a minimum threshold $\mathrm{min_{thr}}$ and it is held until level_1 goes above the half $\mathrm{M}/2$; drain_2 is performed whenever level_2 goes above a maximum threshold $\mathrm{max_{thr}}$; finally, a transfer is decided whenever level_1 is above $\mathrm{max_{thr}}$ or level_2 is below the $\mathrm{min_{thr}}$. This behavior is formalized by formulas (4–9) in Table 2.

$$\text{at 0:} \quad \mathsf{level}_i = \mathrm{M}/2 \tag{1}$$

$$\forall f, l, d : [0..1], L : [0..\mathrm{M}] : \left(\begin{array}{c} \mathsf{Y}(\mathsf{fill}_i = f \wedge \mathsf{leak}_i = l \wedge \mathsf{drain}_i = d \wedge \mathsf{level}_i = L) \\ \Rightarrow \\ \mathsf{level}_i = \min(\mathrm{M}, \max(0, L + f \cdot \mathrm{fr}_i + l \cdot \mathrm{lr}_i + d \cdot \mathrm{dr}_i)) \end{array} \right) \tag{2}$$

$$(\mathsf{trans} = 1 \Rightarrow \mathsf{drain}_1 = \mathsf{fill}_2 = 1) \wedge (\mathsf{trans} = 0 \Rightarrow \mathsf{drain}_1 = \mathsf{fill}_2 = 0) \tag{3}$$

$$\text{at 0:} \quad \mathsf{fill}_1 = \mathsf{trans} = \mathsf{drain}_2 = 0 \tag{4}$$

$$\mathsf{level}_1 \leq \mathrm{min_{thr}} \quad \Rightarrow \quad \mathsf{fill}_1 = 1 \tag{5}$$

$$\mathsf{fill}_1 = 1 \quad \Rightarrow \quad \mathsf{level}_1 \leq \mathrm{M}/2 \tag{6}$$

$$\mathsf{Y}(\mathsf{fill}_1 = 0) \wedge \mathsf{fill}_1 = 1 \quad \Rightarrow \quad \mathsf{U}\,(\mathsf{fill}_1 = 1, \mathsf{level}_1 \geq \mathrm{M}/2) \tag{7}$$

$$\mathsf{level}_1 \geq \mathrm{max_{thr}} \vee \mathsf{level}_2 \leq \mathrm{min_{thr}} \quad \Leftrightarrow \quad \mathsf{trans} = 1 \tag{8}$$

$$\mathsf{level}_2 \geq \mathrm{max_{thr}} \quad \Leftrightarrow \quad \mathsf{drain}_2 = 1 \tag{9}$$

$$1 \leq \mathsf{level}_1 \leq \mathrm{M} - 1 \quad \wedge \quad 1 \leq \mathsf{level}_2 \leq \mathrm{M} - 1 \tag{10}$$

$$\mathsf{level}_1 = \mathrm{min_{thr}} \quad \Rightarrow \quad \mathsf{F}_{[1..\mathrm{M}/2 - \mathrm{min_{thr}}]}(\mathsf{level}_1 \geq \mathrm{M}/2) \tag{11}$$

Table 2. Formulas of the Reservoir System.

We verify that, under suitable choices for the parameters, the following two properties hold for the modular system: the level of both reservoirs is always in the range $[1..\mathrm{M}-1]$ (formula (10) in Table 2), and if level_1 reaches minimum threshold $\mathrm{min_{thr}}$ it grows back to the value $\mathrm{M}/2$ in no more than $\mathrm{M}/2 - \mathrm{min_{thr}}$ time units (formula (11) in Table 2). For all formulas, except (1) and (4), we assume an implicit quantification over the whole temporal axis with the Alw operator.

System verification. In order to evaluate the effectiveness of our approach we verified the reservoir system using both the flat "vanilla" approach — as presented in [13] — and the modular approach of this paper. The model for the flat verification can be automatically generated using the TRIO2ProMeLa translator.[6] In a nutshell, TRIO2ProMeLa translates MTL (and TRIO) formulas in ProMeLa models, the input language of the Spin model-checker [9]. The generated ProMeLa models simulate alternating automata, which are finite-state automata over infinite computations, potentially exponentially more concise variants of Büchi automata [18]. The ProMeLa simulation

[6] TRIO2ProMeLa is available at http://home.dei.polimi.it/spoleti/TRIO2ProMeLa.htm, together with the code used in the experiments.

accepts (or rejects) computations by analyzing the validity of the current value of variables at each step, also taking in account the current history of the computation. In the flat approach, the modular structure of the system is ignored, hence computations are generated by a unique global generator that proceeds exhaustively step by step, until acceptance or rejection can be decided.

In the modular setting, we translated MTL formulas similarly as in the flat case but we associated different ProMeLa processes to each system module. Each process is represented in ProMeLa using a `proctype` instance and the order among them is enforced through message passing, so that each process generates only the variables needed at that point of the analysis.

Table 3 shows several test results obtained by running modified TRIO2ProMeLa models of the reservoir system described above with the Spin model-checker. In all tests we assume: $\min_{\mathrm{thr}} = 5, \max_{\mathrm{thr}} = \mathrm{M} - 2, \mathrm{fr}_1 = 4, \mathrm{dr}_1 = -\mathrm{fr}_2 = -2, \mathrm{lr}_1 = \mathrm{lr}_2 = -1, \mathrm{dr}_2 = -2$. For each test the table reports: whether a flat or modular model is used (F/M), the value of parameter M in the specification, the checked property, the total ordering of modules according to which variables are generated, if some additional modification are introduced in the model (to be discussed next), the number of explored states and transitions (in millions), the used memory (in MBytes, ∞ means "out of memory"), and the verification time (in seconds). The tests have been performed on a PC equipped with an AMD Athlon64 X2 Dual-Core Processor 4000+, 2 GBytes of RAM (roughly 1.7 Gb available), Kubuntu GNU/Linux (kernel 2.6.24), Spin 5.1.5 (using partial order reduction and memory compression).

The experiments show clearly that the reservoir system cannot be analyzed with the flat approach, and taking into account the modular structure is needed. The connection graph of the reservoir system is cyclic and its FASs with a minimum number of arcs are $F_1 = \{\mathsf{level}_1, \mathsf{level}_2\}$, $F_2 = \{\mathsf{fill}_1, \mathsf{drain}_1, \mathsf{fill}_2, \mathsf{drain}_2\}$, $F_3 = \{\mathsf{level}_1, \mathsf{drain}_1, \mathsf{fill}_2, \mathsf{level}_2\}$, and $F_4 = \{\mathsf{fill}_1, \mathsf{trans}, \mathsf{drain}_2\}$. According to the notion of weighted connection graph introduced beforehand, FASs including level_1 and level_2 have the highest weight and thus are likely to be inconvenient. Correspondingly, the two best FASs are F_2 and F_4, which yield the generation orderings $G_1 = \mathsf{Reservoir}_1, \mathsf{Reservoir}_2, \mathsf{Control}, \mathsf{Pipe}$ and $G_2 = \mathsf{Pipe}, \mathsf{Reservoir}_1, \mathsf{Reservoir}_2, \mathsf{Control}$, respectively.[7] Among them G_2 is the best one according to the weighted connection graph heuristics. Experiments confirm that these two orderings yield the best performances; however, G_1 is slightly better than the other ordering G_2. This is acceptable, given that the corresponding FASs have a very similar weight. The chosen ordering G_1 also allows us to scale nicely verification parameter M up to a value of 31. In addition, the G_1 and G_2 orderings also correspond to the lowest GDD values, where in particular G_1 gets the best score in this case. This gives support to the intuition that the Reservoir modules "filter out" more than Pipe and hence they should come first in the ordering.

In addition to the modular technique, in the experimentation we also tinkered with some *ad hoc* optimizations. First, we noticed that $\mathsf{Reservoir}_1$ and $\mathsf{Reservoir}_2$ use only past values of the outputs provided by Control and Pipe. Hence, we avoided generating inputs to $\mathsf{Reservoir}_1$ and $\mathsf{Reservoir}_2$ (when these modules come first in the ordering), as their inputs are already provided by generation in other modules in previous steps.

[7] For symmetry, we fix the relative order between modules R_1 and R_2.

This corresponds to optimization (A) in Table 3. An additional consideration is that Pipe only contains "definition" formulas, composed by present and past modalities. So the input values to be considered can be immediately narrowed down only to the two cases $\mathsf{trans} = \mathsf{drain}_1 = \mathsf{fill}_2 = 0$ and $\mathsf{trans} = \mathsf{drain}_1 = \mathsf{fill}_2 = 1$. This corresponds to optimization (B) in Table 3. Optimization (C) combines optimizations (A) and (B). Finally, optimization (D) further extends the idea of optimization (B) by noticing that all formulas except (7) do not involve future modalities and hence the corresponding variable generation can be done deterministically.

F/M	M	Prop.	Gen. order	Notes	MStates	MTrans.	Mem.	Time
F	12	(10)			34.00	36.92	∞	165
F	12	(11)			34.00	36.92	∞	166
M	12	(10)	PCR_1R_2		11.34	12.03	632	75
M	12	(11)	PCR_1R_2		11.34	12.03	632	75
M	12	(10)	PR_1R_2C		1.81	1.95	94	11
M	12	(11)	PR_1R_2C		1.81	1.95	94	10
M	12	(10)	R_1R_2CP		0.53	0.58	28	3
M	12	(11)	R_1R_2CP		0.53	0.58	28	3
M	12	(10)	CPR_1R_2		16.04	17.11	896	105
M	12	(11)	CPR_1R_2		16.04	17.11	896	103
M	20	(10)	PR_1R_2C		6.43	6.92	357	42
M	25	(10)	PR_1R_2C		13.31	14.33	773	90
M	31	(10)	PR_1R_2C		29.42	31.68	1667	193
M	32	(10)	PR_1R_2C		29.43	31.70	∞	194
M	12	(10)	R_1R_2CP	A	0.05	0.06	3	0
M	12	(11)	R_1R_2CP	A	0.05	0.06	3	0
M	12	(10)	R_1R_2CP	B	0.53	0.57	28	3
M	12	(11)	R_1R_2CP	B	0.53	0.57	28	3
M	12	(10)	R_1R_2CP	C	0.05	0.05	3	0
M	12	(11)	R_1R_2CP	C	0.05	0.05	3	0
M	100	(10)	PR_1R_2C	C	2.11	2.29	110	13
M	150	(10)	PR_1R_2C	C	3.12	3.39	172	21
M	12	(10)	PR_1R_2C	D	0.13	0.16	7	1
M	12	(11)	PR_1R_2C	D	0.13	0.16	7	1
M	100	(10)	PR_1R_2C	D	0.05	0.05	3	0
M	150	(10)	PR_1R_2C	D	0.07	0.07	4	0

Table 3. Experiments with the Reservoir System.

5 Conclusion

We showed how the information on the modular structure of composite systems can be availed to increase the efficiency of the state-space exploration in explicit-state linear-time model-checking. We introduced heuristic techniques that extract a total ordering among modules of a complex system according to its topology. Experiments have shown clearly very relevant performance enhancements when the state-space exploration is done according to the technique. In particular, the verification of the example system has been made possible with limited resources.

Future work will follow four main directions. First, we are going to implement our exploration technique in the TRIO2ProMeLa translator (which can currently handle flat specifications only). Second, we will consider additional optimization techniques that

can be useful in the modular case, such as those taking into account information about the modules' semantics and their capability of "filtering out" unproductive variable values. Third, we plan to consider the problem of applying our approach to systems with heterogeneous behavioral specifications of modules — such as different kinds of automata and temporal logics — possibly considering additional *ad hoc* optimizations for significant special cases. Fourth, we will investigate if techniques similar to those presented in this paper can be used effectively also with BDD-based model-checking techniques.

Acknowledgements. We thank the anonymous reviewers for their useful remarks.

References

1. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
3. W. P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, 1998.
4. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072. Elsevier Science, 1990.
5. C. A. Furia. A compositional world: a survey of recent works on compositionality in formal methods. Technical Report 2005.22, DEI, Politecnico di Milano, 2005.
6. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
7. M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1993.
8. D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498, 1985.
9. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. 2003.
10. R. M. Karp. Reducibility among combinatorial problems. In *Proceedings of the Symposium on Complexity of Computer Computations*, pages 85–103, 1972.
11. O. Kupferman and M. Y. Vardi. An automata-theortetic approach to modular model checking. *ACM TOPLAS*, 22(1):87–128, 2000.
12. O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.
13. A. Morzenti, M. Pradella, P. San Pietro, and P. Spoletini. Model-checking TRIO specifications in SPIN. In *FME 2003*, volume 2805 of *LNCS*, pages 542–561, 2003.
14. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
15. M. Pradella, P. San Pietro, P. Spoletini, and A. Morzenti. Practical model checking of LTL with past. In *ATVA'03*, pages 135–146, 2003.
16. P. San Pietro, A. Morzenti, and S. Morasca. Generation of execution sequences for modular time critical systems. *IEEE TSE*, 26(2):128–149, 2000.
17. P. Spoletini. *Verification of Temporal Logic Specification via Model Checking*. PhD thesis, DEI, Politecnico di Milano, May 2005.
18. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–164. Elsevier Science, 1990.