libFAUDES – An Open Source C++ Library for Discrete Event Systems

Thomas Moor*, Klaus Schmidt*, Sebastian Perk*

* Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg, D-91058 Erlangen, Germany
{thomas.moor,klaus.schmidt,sebastian.perk}@rt.eei.uni-erlangen.de

***Abstract*— The libFAUDES (Friedrich-Alexander University Discrete Event Systems) library is an open source C++ software library for discrete event systems (DES) that is developed at the University of Erlangen-Nuremberg. The core library supports methods for the DES analysis and supervisor synthesis, while a built-in plugin mechanism allows of specialized library extensions. In this paper, we evaluate libFAUDES according to the benchmark examples provided for the Workshop on Discrete Event Systems 2008.**

I. Introduction

The publicly available C++ software library libFAUDES (Friedrich-Alexander University Discrete Event Systems library) [1] has been developed at the Chair of Automatic Control, University of Erlangen-Nuremberg. The core library provides basic algorithms for the manipulation of finite state automata and the supervisor synthesis in the Ramadge/Wonham framework [2]. A plugin system facilitates extensions of the core library such as the methods for timed automata [3], I/O-based control [4] and hierarchical control [5] that have already been implemented.

In this paper, we evaluate the performance of libFAUDES by means of the WODES 2008 benchmark examples [6]. In addition to the monolithic supervisor synthesis, we apply *locally modular control* [7] and *hierarchical and decentralized control* [5] to demonstrate the practical use of our library.

The remainder of the paper is organized as follows. In Section II, we describe the features of the libFAUDES software library and its use for the benchmarking process. The evaluation of the "dining philosophers" example and the "cat-and-mouse" example with monolithic supervisory control is presented in Section III. Section IV provides our computational results for the application of locally modular control and hierarchical control to the benchmark problems, and we give conclusions in Section V.

II. The libFAUDES Software Library

A. libFAUDES

The libFAUDES library [1] provides a C++ environment for the implementation of algorithms for the analysis and synthesis of discrete event control systems. While we use Linux as our primary development platform, libFAUDES compiles flawlessly under Microsoft Windows and Apple Mac OsX. The initial version was written by Bernd Opitz as a masters project [8], and since then has been further developed by numerous students and the authors of this paper.

Technically, libFAUDES is built on top of the C++ Standard Template Library (STL) to define container classes for event sets, state sets and transition relations. When compared to plain C arrays, using STL does introduce some overhead regarding memory usage. However, STL provides a rich set of efficient basic algorithms (searching, sorting, etc), and we feel that this is a worthwhile trade off. All libFAUDES containers take an attribute template parameter to support user defined attributes. For example, an event set may use the attribute parameter to indicate whether individual events are controllable and/or observable.

For an application interface, the libFAUDES *generator class* models *finite automata* and provides access to states, events and transitions via indices, symbolic names and iterators as well as a human readable file IO. The library also implements a variety of general purpose operations on generators and the respective formal languages (union, intersection, Kleene closure, projection, minimal state realization, etc.). We include basic functions for supervisory controller synthesis to demonstrate how the library can serve as an environment for implementation and evaluation of algorithms for the analysis and synthesis of discrete event systems.

libFAUDES is distributed as open source in the hope that it may be useful for researchers in the DES field. We are committed to the further development and welcome suggestions and contributions. We chose the GNU Lesser General Public License (LGPL [9]) scheme since this protects external contributions while it imposes no restrictions on libFAUDES based applications. In particular, there is no requirement to make the application open source. Furthermore, the build system supports a plugin mechanism to strictly separate the core library from library extensions. Authors are invited but not forced to contribute their algorithms to the core library.

B. luaFAUDES

The FAUDES library is supplemented by the console application luaFAUDES. It is based on the scripting language Lua [10] and provides a comfortable console environment to execute libFAUDES functions. All benchmark experiments in this paper have been coded as luaFAUDES scripts and can be found on the libFAUDES homepage [1]. The below script illustrates an example monolithic supervisor synthesis for a plant consisting of four components with associated specifications. This script may be typed (and executed) line-by-line in the luaFAUDES console; alternatively, it can be stored in a file and executed by luaFAUDES as a Lua script.

```
--read data of first plant/specification
--from .gen-file
plant = Generator.New("plant1.gen")
spec  = Generator.New("spec1.gen")

--compose with plants/specifications 2-4
for i = 2,4 do
 iplant = Generator.New(
   string.format("plant%d.gen",i))
 ispec  = Generator.New(
   string.format("spec%d.gen",i))
 Parallel(plant,iplant,plant)
 Parallel(spec,ispec,spec)
end

--synthesize nonblocking supervisor
super = Generator.New()
SupConNB(plant,spec,super)

--report statistics of result to console
super:StatWrite()

% Statistics for SupConNB(plant, spec)
% States:        362
% Init/Marked:   1/1
% Events:        68
% Transitions:   1159
% StateSymbols:  362
% Attrib. E/S/T: 60/0/0
```

C. Further development and Applications

There is a number of libFAUDES applications and add-ons in different stages of development, partly conducted as student projects.

- Functions to support the *hierarchical synthesis methods* of the authors [4], [5], [11], [12] are implemented as the libFAUDES plugins "hiosys" and "schmidt". Routines of general interest will be integrated in the library core in the near future.
- A *code generator* that translates libFAUDES generators into IEC 61131-3 code has been developed in collaboration with Infoteam Software GmbH [13]. It includes the libFAUDES plugin "timed" for timed automata [3] using set attributes to model time constraints.
- A *simulator* for synchronized timed generators has been developed as a libFAUDES application [14]. It supports both the interactive and the stochastic synchronous execution of timed generators.
- A *device driver intermediate layer* has been designed to map physical sensor and actuator signals from digital IO hardware to libFAUDES events to enable hardware-in-the-loop simulation.
- A Qt [15] widget for the graphical representation of generators has been developed as a basis for a *graphical user interface* to libFAUDES.

D. Benchmark Experiments

For the respective WODES'08 benchmark examples, the libFAUDES plugins "philosophers" and "cats-and-mice" have been developed. The plugins compile to several binaries (or executables). Each binary represents one of the approaches in Sections III-IV and performs the tasks announced on the WODES'08 website [6]:

- construct plant model(s) and specification(s) according to the command line options n and k and design nonblocking, maximally permissive supervisor(s)
- output statistical data to the console
- output supervisor(s) to human readable files

The benchmark experiments were carried out under Linux, and a shell script was used to execute the respective binary looping over n and k while measuring the values TIME and MEMORY for each run. For the value TIME, we chose the *total time* that elapsed from starting the process that executes the binary to completing the file output. The value MEMORY represents the *maximum* amount of *physical memory* used by the process, which comprises the memory consumed by the executed code as well as the data generated during the process. For comparability of all benchmark experiments and to avoid swapping to virtual memory, *all runs exceeding 1 GB memory were regarded unsuccessful and terminated*. The benchmark experiments where performed on customary PC's with 3 GHz CPU's and 3 GB physical memory.

The computational results and measurement values obtained for the approaches presented in the following sections are summarized in Tables I - VI. The entries for each pair (n, k) contain three values: The above-mentioned values MEMORY and TIME are denoted by M and T, respectively. Besides the unit *second* (s), we also used the units *millisecond* (ms), *minute* (min), *hour* (h) and *day* (d) for the TIME value. In Tables I - III, the value S denotes the number of states of the closed loop that serves as a realization of the supervisor; in Tables IV - VI, S indicates the *sum of states* of all resulting supervisors.

Besides some representative values for n and k, the tables also contain the *highest* values achieved using less than 1GB memory. Higher values for n and k, e.g., entries filled with "—", exceed this memory limit.

III. Benchmark Evaluation: Monolithic Supervisor Synthesis

In this section, we apply the monolithic supervisor synthesis to both the "dining philosophers" and the "cat-and-mouse" benchmark example. To this end, we first determine an overall plant automaton G and an overall specification automaton D with $L_{\mathrm{m}}(D) \subseteq L_{\mathrm{m}}(G)$. Then, the libFAUDES routine SupConNB is invoked to synthesize the *nonblocking* and *maximally permissive* supervisor automaton R, i.e.,

$$R = \mathsf{SupConNB}(G, D). \tag{1}$$

A. Dining philosophers Example

The "dining philosophers" benchmark example (cf. Example 2.17 in [16]) considers $n \geq 2$ philosophers $P_1, \ldots, P_n$ sitting around a table such that philosopher P_n is a neighbor of philosopher P_1. There is one fork $F_{i1,i2}$ between any two philosophers P_{i1} and P_{i2} representing a shared resource. A philosopher i can pick up the left fork first (event ifl), then the right fork (event ifr), and then drop both

TABLE I

MONOLITHIC APPROACH: DINING PHILOSOPHERS EXAMPLE

$n \rightarrow$		2	3	4	5	6	7	8	9	10	11	12	13
k	M	<1MB	<1MB	1.2MB	1.2MB	1.3MB	1.6MB	2.3MB	3.8MB	8.4MB	19MB	48MB	122MB
=	T	<1ms	<1ms	<1ms	1ms	2ms	10ms	25ms	66ms	2s	6s	19s	75s
1	S	3	12	30	78	190	470	1138	2770	6694	16 206	39 138	94 578
k	M	1.2MB	1.2MB	1.3MB	1.5MB	2.4MB	4.7MB	141MB	47MB	166MB	—	—	—
=	T	<1ms	<1ms	1ms	6ms	1s	1s	9s	65s	643s	—	—	—
2	S	4	24	83	321	1082	3855	12 863	44 172	146 675	—	—	—
k	M	1.2MB	1.2MB	1.4MB	2.1MB	5.5MB	22MB	99MB	—	—	—	—	—
=	T	<1ms	<1ms	3ms	19ms	2s	38s	724s	—	—	—	—	—
3	S	5	40	178	932	4050	19 292	82 946	—	—	—	—	—
k	M	1.2MB	1.7MB	7.5MB	75MB	855MB	—	—	—	—	—	—	—
=	T	<1ms	8ms	5s	1120s	1d13h	—	—	—	—	—	—	—
9	S	11	220	2302	32 150	331 902	—	—	—	—	—	—	—
k	M	1.3MB	3.4MB	42MB	774MB	—	—	—	—	—	—	—	—
=	T	<1ms	70ms	350s	1d9h	—	—	—	—	—	—	—	—
15	S	17	544	8962	205 136	—	—	—	—	—	—	—	—
k	M	846MB	—	—	—	—	—	—	—	—	—	—	—
=	T	1d8h	—	—	—	—	—	—	—	—	—	—	—
1436	S	1438	—	—	—	—	—	—	—	—	—	—	—

forks (event if). The parameter k means that by picking up the left fork, a philosopher i reaches the first of k intermediate states, conducts $k-1$ intermediate transitions (events $it1,..,itk-1$) to reach intermediate state k, where the right fork is picked up. For the case $n = k = 2$, the philosopher model P_2 and the model $F_{1,2}$ of the fork between P_1 and P_2 are depicted in Fig. 1. The uncontrollable events are ifl, for i even. The monolithic plant model is $G := F_{n,1}||P_1||(||_{i=2}^{n}(F_{i-1,i}||P_i))$. A deadlock situation occurs, when each philosopher holds the left fork.

With the trivial specification $D := G$, the maximally permissive and nonblocking supervisor R is computed according to (1) for different values of n and k as depicted in Table I. It can be seen that the computation is limited by the value of $n = 13$ for $k = 1$ and $k = 1436$ for $n = 2$.

B. Cat-and-Mouse Example

Our study is based on the "cat-and-mouse" example initially described in [2]. We investigate two different models for this example with its extension to n maze levels and k cats and mice. In the first case, we provide individual models for each cat and mouse, while the second model does not distinguish between different cats and mice. It rather captures the number of cats and mice in each room of the maze.

1) Individual Models for Cats and Mice: We model the i-th cat by an automaton G_i and the i-th mouse by an automaton G_{k+i}, where $1 \leq i \leq k$, such that each automaton G_i describes the behavior of the respective animal on all n levels of the maze. As an example, the automaton G_1 for the 1-st cat on 2 maze levels is shown in Fig. 2. Here, each event `cx-y(l,1)` indicates that the cat 1 moves from room x to room y on the maze level l, while `cz(l,1)` describes the movement of cat 1 from level l to level z. Similarly, the model G_{k+i} for the i-th mouse is determined. The overall plant is $G := ||_{i=1}^{2k} G_i$.

Furthermore, it has to be specified that if a cat stays in a certain room of the maze, no mouse is allowed to enter and vice versa. We characterize this restriction by one specification D_i, $1 \leq i \leq k$ for the i-th cat and one specification D_{k+i}, $1 \leq i \leq k$ for the i-th mouse. For the i-th cat, the specification automaton D_i is derived from the plant automaton G_i as follows: in each state of G_i, all mouse events are added as a selfloop except for the mouse events entering the corresponding room. For the state 4_1 of the cat specification D_1, this means that all mouse events except for `m0-4(1,1)`,...,`m0-4(1,k)` appear in a selfloop. Using an analogous procedure for the mouse specifications, the overall specification evaluates to $D := ||_{i=1}^{2k} D_i$.

Based on G and D derived as above, the maximally permissive and nonblocking supervisor R is computed according to (1) for different values of n and k as depicted in Table II. It can be observed that the memory consumption grows rapidly with increasing k. Together, with the memory limit of 1 GB, no values for $n > 49$ and $k > 4$ could be obtained.

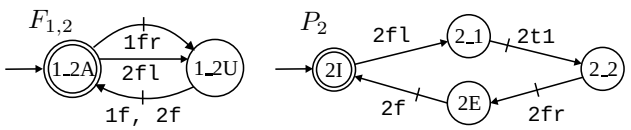


Fig. 1. Fork model $F_{1,2}$ and philosopher model P_2 for $n = k = 2$

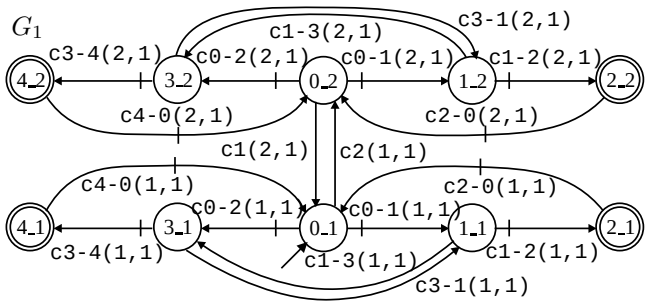


Fig. 2. Individual cat model G_1 for cat 1 on two maze levels

TABLE II

MONOLITHIC APPROACH: INDIVIDUAL CATS AND MICE

$n \rightarrow$		1	2	3	4	5	49
k	M	<1MB	1.3MB	1.6MB	2.0MB	2.6MB	955MB
=	T	<1ms	1ms	3ms	5ms	12ms	57s
1	S	6	82	199	362	582	59609
k	M	1.6MB	13MB	83MB	379MB	—	—
=	T	2ms	1s	44s	474s	—	—
2	S	24	4594	31247	108302	—	—
k	M	10MB	—	—	—	—	—
=	T	31ms	—	—	—	—	—
3	S	90	—	—	—	—	—
k	M	284MB	—	—	—	—	—
=	T	11s	—	—	—	—	—
4	S	336	—	—	—	—	—

2) Individual Models for Maze Levels: We derive n plants $G_1, \ldots, G_n$ for the different maze levels. On each level l, $1 \leq l \leq n$, there is a model $C_l^{k,i}$ for each room $i \in \{1, \ldots, 5\}$ with maximally k cats. In the initial state, each room except for room 0 on level 1 is empty. As an example, Fig. 3 (a) shows the model for room 2 on level 1 of the maze with $k = 2$ cats. Analogously, models $M_l^{k,i}$ for the mice can be determined.

In addition, *cat counters* $\hat{C}_l^k$ and *mouse counters* $\hat{M}_l^k$, $1 \leq l \leq n$, capture the number of cats and mice that can be present in each level l of the maze, respectively. They observe the changes between levels and the initial number of k cats in level 1, k mice in level n, and 0 cats and mice in the remaining levels of the maze. The cat counter $\hat{C}_1^2$ for 2 cats and level 1 is shown in Fig. 3 (b). The events `c2(1)` and `c1(2)` describe the transition of a cat from level 1 to 2 and from level 2 to 1, respectively.

The model G_l for each maze level is hence

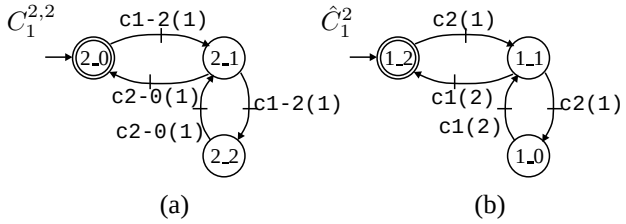
$$G_l := \hat{C}_l^k || \hat{M}_l^k || (||_{i=1}^5 (C_l^{k,i} || M_l^{k,i})), \tag{2}$$

and the overall plant model is $G := ||_{l=1}^n G_l$. Additionally, similar to the construction in the previous section, a specification D_l that forbids cats and mice to enter the same room is determined for each maze level l, $1 \leq l \leq n$. Hence, we arrive at the overall specification $D := ||_{l=1}^n D_l$.

TABLE III

MONOLITHIC APPROACH: INDIVIDUAL MAZE LEVELS

$n \rightarrow$		1	2	3	4	6	59
k	M	1.2MB	1.8MB	1.9MB	1.9MB	2.5MB	946MB
=	T	<1ms	6ms	9ms	11ms	21ms	461min
1	S	6	82	199	362	845	86 525
k	M	5.2MB	43MB	43MB	47MB	310MB	—
=	T	17ms	3s	6s	45s	840s	—
2	S	15	1 506	9 184	30 449	170 036	—
k	M	42MB	788MB	807MB	—	—	—
=	T	2s	68s	3720s	—	—	—
3	S	29	12 446	184 052	—	—	—
k	M	254MB	—	—	—	—	—
=	T	11s	—	—	—	—	—
4	S	49	—	—	—	—	—
k	M	512MB	—	—	—	—	—
=	T	27s	—	—	—	—	—
9	S	274	—	—	—	—	—



(a) (b)

Fig. 3. (a) 2 cats in room 2 of level 1; (b) Cat counter for level 1

The results of the monolithic supervisor computation in (1) for different values of n and k are shown in Table III. Here, the maximum values for $n = 59$ and $k = 9$ are considerably larger than the corresponding values in Table II just because of the different modeling paradigm.

IV. BENCHMARK EVALUATION: METHODS USING SYSTEM STRUCTURE

In this section, we employ the routines of libFAUDES to implement *locally modular control* [7] and *hierarchical and decentralized control* [5], [17], [18]. In particular, we investigate how the use of structural information about the given system affects the applicability of supervisory control.

A. Locally Modular Control

Locally modular control as introduced in [7], assumes that the plant is composed of s subsystems $G_1, \ldots, G_s$ with disjoint alphabets, i.e., $\Sigma_{G_i} \cap \Sigma_{G_j} = \emptyset$ for $i, j \in \{1, \ldots, s\}$ and $i \neq j$. Furthermore, there are m local specification automata $D_1, \ldots, D_m$. Accordingly, m modular plants $H_1, \ldots, H_m$ are constructed from the subsystems that share events with the respective specification such that

$$H_i := \mathop{||}_{j, \Sigma_{G_j} \cap \Sigma_{D_i} \neq \emptyset} G_j \qquad \text{for } i = 1, \ldots, m. \tag{3}$$

For H_i and D_i, m local supervisors are computed using (1):

$$R_i = \texttt{SupConNB}(H_i, D_i) \qquad \text{for } i = 1, \ldots, m. \tag{4}$$

Finally, it has to be verified if the supervisors R_i are nonconflicting, i.e., if $R := ||_{i=1}^m R_i$ is nonblocking. If R is blocking, a further nonblocking overall supervisor has to be computed for the plant R. As has been shown in [7], this method results in a maximally permissive and nonblocking supervisor. The potential benefits of the approach arise from the local evaluation of the supervisor synthesis in (4). Even if an overall supervisor has to be computed, the prior local synthesis can lead to smaller plant models.

Locally modular control cannot be applied to the "dining philosophers" example in Section III-A and the "cat-and-mouse" example with individual models for maze levels in Section III-B.2 as in both cases, the subplants share events.

However, the "cat-and-mouse" example with individual cat and mouse models in Section III-B.1 is suitable for locally modular control. The subplants are given by $G_1, \ldots, G_{2k}$, and the cat and mouse specifications are $D_1, \ldots, D_{2k}$. According to (4), the $2k$ modular plants evaluate to

$$H_i = \begin{cases} G_i || (||_{j=k+1}^{2k} G_j) & \text{for } i = 1, \ldots, k \\ G_i || (||_{j=1}^{k} G_j) & \text{for } i = k+1, \ldots, 2k. \end{cases}$$

The computation of the modular plants and the subsequent local supervisor computation according to (4) has been implemented in libFAUDES. Our routine also includes the verification of nonconflict and/or the synthesis of an overall supervisor based on the modular closed-loops R_i, $i = 1, \ldots, 2k$. Table IV depicts the computational results.

TABLE IV
LOCALLY MODULAR APPROACH: CAT-AND-MOUSE EXAMPLE

$n \rightarrow$		1	2	3	4	49	55
k	M	1.2MB	1.4MB	1.7MB	2.0MB	679MB	901MB
=	T	<1ms	1ms	6ms	11ms	52s	901s
1	S	6	82	199	362	59 609	75 162
k	M	1.5MB	9.2MB	47MB	191MB	—	—
=	T	4ms	1s	12s	125s	—	—
2	S	24	4 594	31 247	108 302	—	—
k	M	4.4MB	706MB	—	—	—	—
=	T	42ms	1 198s	—	—	—	—
3	S	90	185 722	—	—	—	—
k	M	60MB	—	—	—	—	—
=	T	6s	—	—	—	—	—
4	S	336	—	—	—	—	—

The main advantage of the locally modular approach compared to the monolithic approach in Section III-B.1 is the reduced memory consumption. As a result, the supervisor synthesis can be carried out for values of up to $n = 55$.

B. Hierarchical and Decentralized Control

Hierarchical and decentralized control as presented in [5], [12], [18] is based on plants that are composed of s subplants $G_1, \ldots, G_s$ that potentially share events, i.e., $\Sigma_{G_i} \cap \Sigma_{G_j} \neq \emptyset$ is permitted for any $i, j \in \{1, \ldots, s\}$. We denote the overall plant as $G := ||_{i=1}^{s} G_i$. The desired system behavior is specified by s local specification automata $D_1, \ldots, D_s$, where $L_{\mathrm{m}}(D_i) \subseteq L_{\mathrm{m}}(G_i)$ for $i = 1, \ldots, s$. Furthermore, there can be a specification D^{hi} with $\Sigma_{D^{\mathrm{hi}}} \subset \Sigma_G$. Hence, D^{hi} serves as a *high-level specification* that only addresses events in a subset of the overall alphabet Σ_G.

First, s local supervisors are computed:

$$R_i := \mathtt{SupConNB}(G_i, D_i) \text{ for } i = 1, \ldots, s. \tag{5}$$

Then, a *high-level alphabet* $\Sigma_{G^{\mathrm{hi}}} \supseteq \Sigma_{D^{\mathrm{hi}}}$ and a natural projection $p^{\mathrm{hi}} : \Sigma_G^* \rightarrow \Sigma_{G^{\mathrm{ni}}}^*$ are introduced in order to determine the *high-level plant* G^{hi} with

$$L(G^{\mathrm{hi}}) := p^{\mathrm{hi}}(\overset{s}{\underset{i=1}{||}} L(R_i)) \text{ and } L_{\mathrm{m}}(G^{\mathrm{hi}}) := p^{\mathrm{hi}}(\overset{s}{\underset{i=1}{||}} L_{\mathrm{m}}(R_i)). \tag{6}$$

According to [19], the natural projection p^{hi} distributes over the parallel composition in (6) if $\Sigma_{G^{\mathrm{hi}}}$ contains all events that are shared among the local supervisors R_i, $i = 1, \ldots, s$, i.e., $\Sigma_{G^{\mathrm{hi}}} \supseteq \Sigma_{\cap} := \bigcup_{i,j \in \{1,\ldots,s\}, i \neq j} (\Sigma_{R_i} \cap \Sigma_{R_j})$. Hence, the abstraction can be computed for each local supervisor to avoid the computation of the overall plant G.

Finally, the *high-level supervisor* $R^{\mathrm{hi}} = \mathtt{SupConNB}(G^{\mathrm{hi}}, D^{\mathrm{hi}})$ ensures that D^{hi} is met. Together, the overall supervisor evaluates to

$$R := R^{\mathrm{hi}} || R_1 || \cdots || R_s. \tag{7}$$

To achieve that R is nonblocking, it is sufficient that p^{hi} is an $L_{\mathrm{m}}(R)$-observer as defined in [20]. In addition, maximal permissive control is ensured if p^{hi} is *locally control consistent* and the subplants G_i, $i = 1, \ldots, s$ are *mutually controllable* [12].

In [12], it is also proved that the hierarchical and decentralized method is applicable to an arbitrary number of hierarchical levels. In Section IV-B.1 and IV-B.2, we construct such a multi-level hierarchy as follows: we group pairs of neighboring abstracted plant models and compute a nonblocking supervisor for the grouped model, which in turn serves as the low-level model for a subsequent application of the hierarchical and decentralized approach. The iterative hierarchical construction terminates with the computation of a single highest-level supervisor. This concept is illustrated in Fig. 4. The resulting overall supervisor is then given by the parallel composition of all supervisors in the hierarchy.

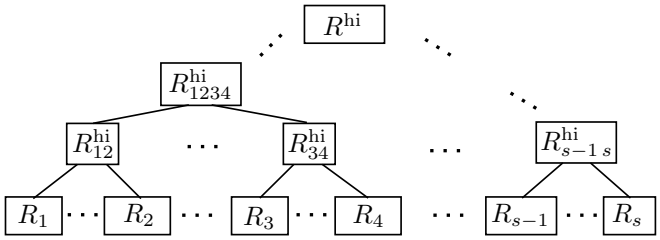


Fig. 4. Hierarchical multilevel construction

The described approach has been implemented in libFAUDES. In order to fulfill the required conditions, our routine verifies mutual controllability, and algorithmically determines a locally control consistent natural projection p^{hi} that satisfies the $L_{\mathrm{m}}(R)$-observer property [12], [18].

There are several potential gains of the approach due to the use of structural system information: first, the overall plant G does not have to be computed. Second, irrelevant behavior is abstracted away before the supervisor R^{hi} is determined. Furthermore, the parallel composition in (7) does not have to be evaluated explicitly such that a set of distributed supervisors on small state spaces can be implemented.

1) Dining Philosophers: Before constructing the supervisor hierarchy of Fig. 4 for the "dining philosophers" problem, mutual controllability of the subplants G_i has to be verified as premise for the overall supervisor to be nonblocking.

It holds that the philosopher models P_i and their respective left forks $F_{i-1,i}$ are *not* mutually controllable for i even! This becomes clear when observing that, e.g., fork $F_{1,2}$ (Fig. 1) can prevent the uncontrollable event $2fl$ in P_2 by being unavailable (i.e. by being in state 1_2U). This problem can be solved by composing each philosopher with its left fork to achieve n mutually controllable local models:

$$G_1 := F_{n,1} || P_1,\ G_2 := F_{1,2} || P_2,\ \ldots,\ G_n := F_{n-1,n} || P_n.$$

With the trivial specifications $D_i := G_i$, the local supervisors in (5) are computed as $R_i = G_i, i = 1, \ldots, n$. Applying the hierarchical construction in Section IV-B without an additional specification D^{hi}, we obtain the following results.

TABLE V

HIERARCHICAL APPROACH: DINING PHILOSOPHERS EXAMPLE

$n \rightarrow$		2	3	5	8	13	16
k	M	1.2MB	1.4MB	1.6MB	1.7MB	27MB	195MB
=	T	$<$ 1ms	2ms	42ms	1s	84min	248min
1	S	15	50	194	646	9 142	87 666

Table V only depicts results for the value $k = 1$ as it turns out that all inserted intermediate states for different values of k are not relevant in the hierarchical synthesis. Hence, all computations for $k > 2$ lead to equivalent results. Comparing Table V with the monolithic evaluation in Table I, it can be seen that there is a potential computational overhead of hierarchical approach for very small values of n (e.g. $n = 2..5$). However, for all larger values of n, the number of supervisor states is significantly smaller (see, e.g., $n = 13$), and supervisors can be synthesized for values up to $n = 16$.

2) Cat and Mouse Example: We consider the "cat-and-mouse" example with individual maze levels as introduced in Section III-B.2. The low-level plant consists of the maze level models $G_1, \ldots, G_n$, where only neighboring maze levels share events. The low-level specifications are given as $D_1, \ldots, D_n$, and there is no additional specification D^{hi}.

The computational results for different values of n and k are shown in Table VI. The benefits of the hierarchical approach compared to the monolithic approach in Section III-B.2 can be seen in both the reduced state counts of the resulting supervisors and the higher maximum values that are achieved for n and k. In particular, for $k = 1$, a tower of up to $21\,384$ maze levels is obtained.

V. CONCLUSIONS

In this paper, the open source C++ library libFAUDES has been presented, and its algorithms have been employed to study the supervisor synthesis problems provided for the benchmark session of the WODES 2008. In addition to the monolithic controller synthesis, *locally modular control* and *hierarchical and decentralized control* have been evaluated in order to demonstrate both the computational benefits of synthesis approaches that respect the system structure and the incorporation of such synthesis approaches in libFAUDES. In particular, the "cat-and-mouse" example could be solved until up to $21\,384$ maze levels.

Future work aims at providing a graphical user interface to the libFAUDES routines.

TABLE VI

HIERARCHICAL APPROACH: CAT-AND-MOUSE EXAMPLE

$n \rightarrow$		2	3	4	17	5000	21 384
k	M	1.6MB	1.7MB	1.8MB	2.0MB	178MB	718MB
=	T	4ms	16ms	16ms	2s	1380s	7d 11h
1	S	54	139	163	1 078	311 884	1 334 229
k	M	4.2MB	44MB	44MB	498MB	—	—
=	T	1s	19s	25s	3h	—	—
2	S	370	1 204	1 516	16 170	—	—
k	M	27MB	792MB	276MB	—	—	—
=	T	10s	900s	3794s	—	—	—
3	S	1 422	7 643	8 031	—	—	—
k	M	467MB	—	—	—	—	—
=	T	1440s	—	—	—	—	—
4	S	4 022	—	—	—	—	—

VI. ACKNOWLEDGMENT

The development of libFAUDES did benefit from contributions made in the context of student projects conducted by (in alphabetical order): Rüdiger Berndt, Christian Breindl, Christoph Dörr, Marc Düvel, Silke Figgen, Norman Franchi, Jochen Hellenschmidt, Mihai Musunoi, Bernd Opitz, Thomas Rempel, Daniel Ritter, Berno Schlein, Johannes Tautz, Thomas Wittmann, Jorgos Zaddach.

REFERENCES

[1] Friedrich-Alexander University Discrete Event Systems library (libFAUDES). [Online]. Available: http://www.rt.eei.uni-erlangen.de/FGdes/faudes/index.php

[2] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings IEEE, Special Issue Discrete Event Dynamic Systems*, vol. 77, pp. 81–98, 1989.

[3] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[4] S. Perk, T. Moor, and K. Schmidt, "Controller synthesis for an I/O-based hierarchical system architecture," *Workshop on Discrete Event Systems (WODES)*, 2008.

[5] K. Schmidt, T. Moor, and S. Perk, "Nonblocking hierarchical control of decentralized discrete event systems," *to appear in IEEE Transactions on Automatic Control*, November 2008.

[6] WODES08. Benchmarking Software Tools. [Online]. Available: http://www.wodes2008.org/pages/benchmark.php

[7] M. H. de Queiroz and J. E. R. Cury, "Modular supervisory control of large scale discrete event systems," *Workshop on Discrete Event Systems*, 2000.

[8] B. Opitz, "Methods of supervisory control: A software implementation," *Master Thesis, Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg*, 2006.

[9] Gnu Lesser General Public License (GLPL). [Online]. Available: http://www.gnu.org/copyleft/lesser.html

[10] The programming language Lua. [Online]. Available: http://www.lua.org/

[11] S. Perk, T. Moor, and K. Schmidt, "Hierarchical discrete event systems with inputs and outputs," *Workshop on Discrete Event Systems (WODES)*, 2006.

[12] K. Schmidt and C. Breindl, "On maximal permissiveness of hierarchical and modular supervisory control approaches for discrete event systems," *Workshop on Discrete Event Systems (WODES)*, 2008.

[13] B. Schlein, "Timed automata as a graphical programming language for the generation of IEC 61131-3 conform PLC code," *Diplomarbeit, Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg*, 2006.

[14] C. Dörr, "Simulation and analysis of discrete event systems," *Diplomarbeit, Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg*, 2008.

[15] Trolltech. Qt: Cross-platform rich client development framework. [Online]. Available: http://www.trolltech.com

[16] C. G. Cassandras and S. Lafortune, "Introduction to discrete event systems," *Kluwer*, 1999.

[17] K. Schmidt. (2005) Hierarchical control of decentralized discrete event systems: Theory and application. Phd-thesis, Lehrstuhl für Regelungstechnik, Universität Erlangen-Nürnberg. [Online]. Available: http://www.rt.eei.uni-erlangen.de/FGdes/dissertation2005_schmidt.pdf

[18] K. Schmidt and T. Moor, "Computation of marked string accepting observers for discrete event systems," *Workshop on Discrete Event Systems (WODES)*, 2006.

[19] K. Schmidt, J. Reger, and T. Moor, "Hierarchical control of structural decentralized DES," *Workshop on Discrete Event Systems*, 2004.

[20] K. C. Wong and W. M. Wonham, "On the computation of observers in discrete-event systems," *Discrete Event Dynamic Systems*, vol. 14, no. 1, pp. 55–107, 2004.