

Model Checking Safety Properties in Modular High-Level Nets

Marko Mäkelä

Laboratory for Theoretical Computer Science

Helsinki University of Technology

P.O.Box 9205

02015 HUT

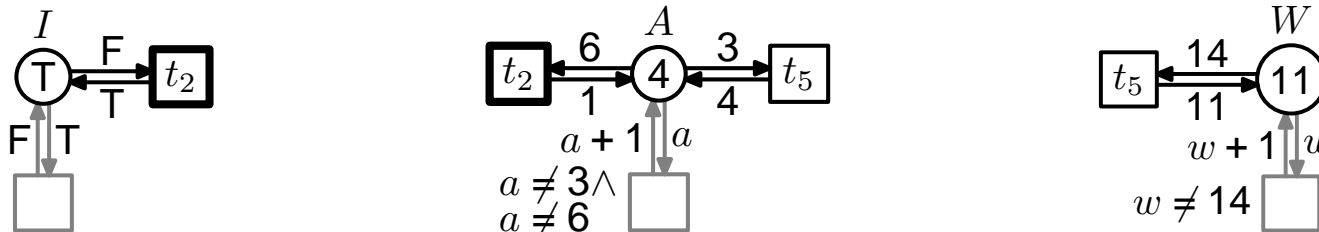
Finland

June 27, 2003

Modular Nets with Shared Transitions

Complex systems tend to consist of loosely connected modules, which may perform internal tasks in parallel.

To efficiently model and analyse such systems with Petri nets, it is useful to extend the nets with modules. In *nested modular nets*, modules communicate via transitions that share a synchronisation label:*

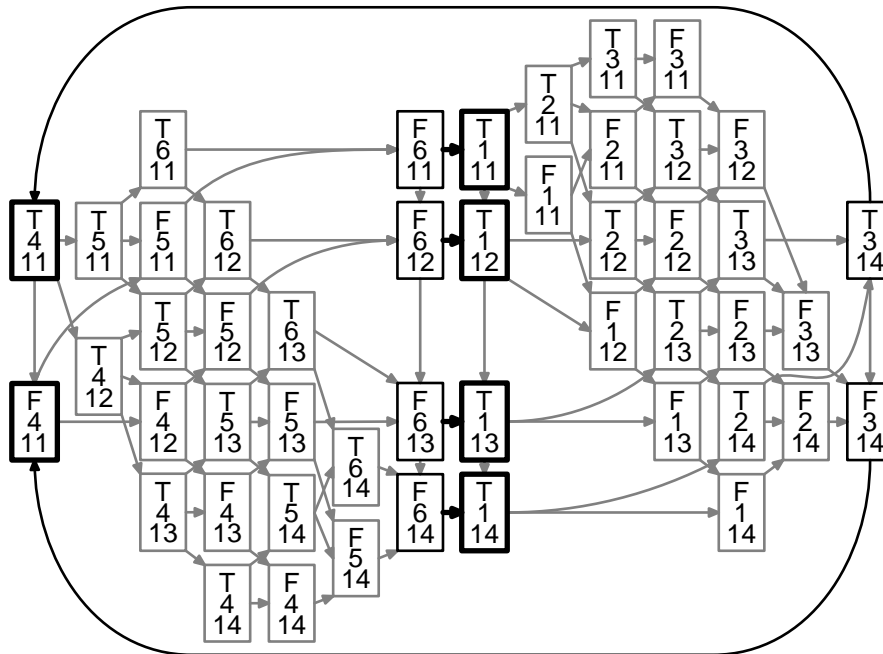


The model above consists of three non-empty modules that synchronise on t_2 and t_5 .

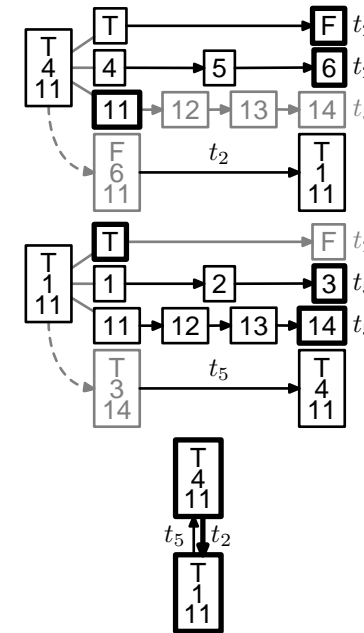
*Christensen and Petrucci have shown how modules with shared places can be converted to this formalism.

Marko Mäkelä

Alleviating the State Explosion with Modular Analysis



flat state space
(48 states, 98 events)



modular state space
(2 sync states, 2 events)

Modular analysis hides states resulting from internal tasks being performed in parallel.

Marko Mäkelä

Nested Modular Nets (1 of 2)

- set of *modules* (high-level nets) S
- *hierarchy tree* $H = (S, C)$ of modules, with a *root* module $s_0 \in S$
- *transition fusion set* TF : each $tf \in TF$ is a set of transitions that may only occur in a synchronous step
- *H-hierarchical TF*: a $tf \in TF$ may only contain transitions of sibling modules $s' \in C(s)$ (sharing a common parent s)
- *internal transitions* do not belong in (synchronise on) any $tf \in TF$

Nested Modular Nets (2 of 2)

Nested modular nets $(H = (S, C), TF, \mathcal{M})$ extend the underlying nets $s \in S$ with *modular augmentations* $\mathcal{M}(s)$:

- The state (marking) of a module $\mathcal{M}(s)$ comprises the places of s and its descendants, $C^+(s)$.
- The state of the root module comprises the places of all modules $s \in S$.
- The marking of a module $\mathcal{M}(s)$ can be projected on its descendants.
- A synchronisation transition is instantiated for each synchronisation label $tf \in TF$ as a fusion of the transitions $t \in tf$.

Occurrence Rule for Nested Modular Nets

The occurrence rule is extended for the synchronisation transitions.

A synchronisation transition tf is $\mathcal{M}(s)$ -enabled in the marking M if

- $M^* = M$ or
- there is a sequence of internal transitions $t_1 \dots t_n$ leading from M to M^* , such that each t_i belongs to some $s' \in C(s)$ that synchronises on tf

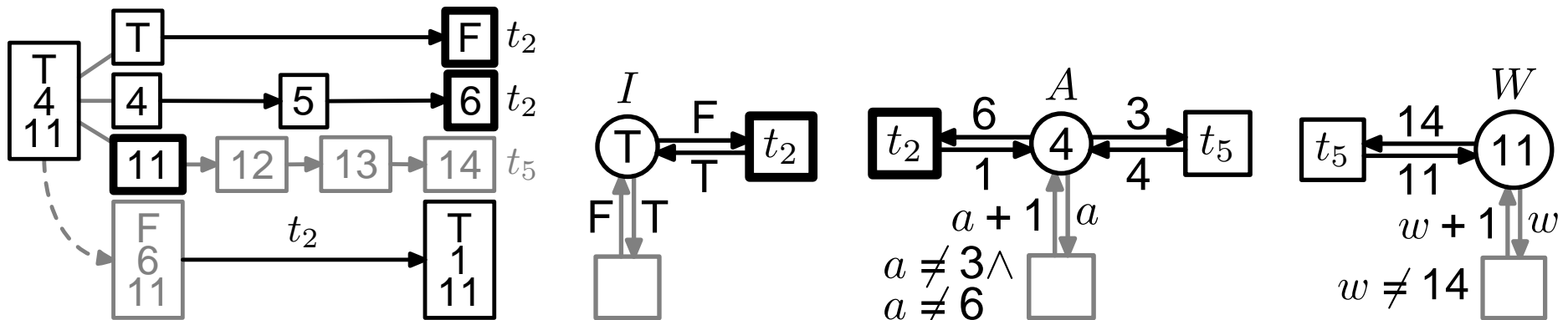
and tf is enabled in the marking M^* .

An $\mathcal{M}(s)$ -enabled transition may $\mathcal{M}(s)$ -occur.

Modular State Spaces

Modular state spaces are obtained by systematically applying the occurrence rule. The main difference from flat state spaces is that a modular system corresponds to multiple state space graphs, one for each module.

In our example, the root module is empty, and its modular augmentation contains two transitions, t_2 and t_5 . In the initial state of the root module, $\{(I, T), (A, 4), (W, 11)\}$, only t_2 is enabled and may occur, producing the marking $\{(I, T), (A, 1), (W, 11)\}$:



Exhaustive Modular State Space Exploration

The recursion in the exploration algorithm supports modules within modules.

The procedure `EXPLORE` constructs the modular state space graph of a given module.

In each reachable state, the procedure `TRANSITIONS` checks whether any external transitions are enabled. The possible synchronisation points are recorded in the relation \mathcal{S} .

This synchronisation relation \mathcal{S} is the input of the procedure `SYNC` that constructs synchronisation states M^* and computes the successor states of the fused synchronisation transitions.

Specifying Safety Properties

Safety properties can be specified as something that can be checked in every reachable local state, such as

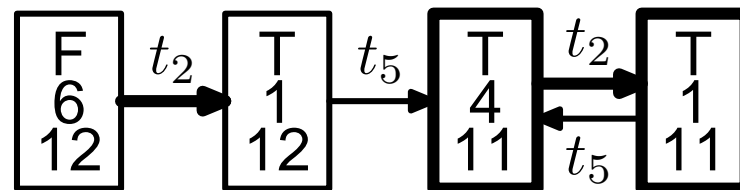
- a “reject” condition,
- a “deadlock” condition (evaluated in states with no enabled transitions), or
- failure to compute a successor state.

More complex safety properties can be stated with property automaton modules.

Experiment: Automated Guided Vehicles

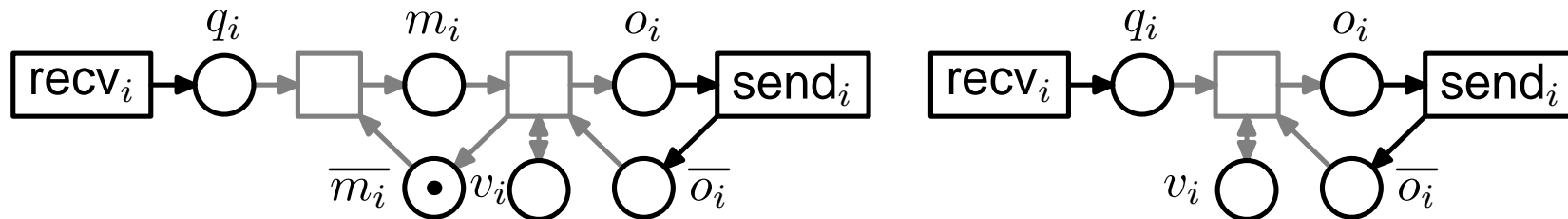
Our example system is part of a larger model, whose flattened state space consists of 30,965,760 nodes and 216,489,984 edges.

The modular state space of the root module of this system comprises 836 nodes and 2,644 edges. There is one strongly connected component of 512 nodes and 1,600 edges. In terms of our example, the state space looks like this:



The state space can be reduced further to 512 nodes and 1,600 edges by “undoing” the effect of internal transitions in the initial marking.

Experiment: Leader Election Protocol



(a) with intermediate buffer

(b) direct consumption

n	flat (a)		flat (b)		modular	
	nodes	edges	nodes	edges	nodes	edges
3	155	299	69	126	33	63
4	712	1,847	240	588	90	227
5	3,428	11,194	870	2,693	251	800
6	16,788	66,039	3,213	12,013	713	2,746
7	82,663	380,263	11,949	52,310	2,041	9,210
8	407,695	2,146,961	44,544	223,338	5,863	24,267

Experiment: Sliding Window Protocol

$tw,$ rw	flat		modular		
	nodes	edges	nodes	edges	(cache)
1,1	108	310	64	172	96
1,2	462	1,686	348	1,734	636
1,3	1,336	5,372	1,104	7,928	2,192
2,1	2,118	8,349	1,422	9,408	2,853
2,2	9,388	40,256	7,080	65,692	15,156
2,3	27,265	122,555	22,115	268,185	49,140
3,1	25,292	113,036	18,412	216,304	40,792
3,2	109,550	508,790	84,775	1,235,830	193,835
3,3	323,724	1,537,638	262,026	4,629,366	611,754

This system is badly suited for modular analysis, because internal actions are rare and a given synchronisation state M^* is reachable from several states M .

Marko Mäkelä

Tuning the Algorithm

Our algorithm recomputes the local state spaces every time a new state is visited in the parent module.

Caching information on possible synchronisation states can significantly speed up the analysis, but it may also blow up the memory usage.

The article describes two caching schemes and reports their effect on the sliding window protocol. We do not think that they are very practical, except perhaps as a benchmark for more sophisticated schemes.

Conclusion and Future Work

Nested modular nets are slightly more general than the modular high-level nets of Christensen and Petrucci.

An modular state space enumeration algorithm was presented. Our implementation consumes only slightly more memory per state than the algorithm for flat state spaces.

We believe that utilising modular analysis can improve productivity:

- less need to “optimise” or prepare models for verification
- structured, unoptimised models may be easier to maintain
- entire models can be reused as modules, e.g., in a multi-layered protocol

The algorithm should be extended to checking liveness properties.

Strategies for decomposing systems into modules should be investigated.

Marko Mäkelä