AVACS – Automatic Verification and Analysis of Complex Systems

REPORTS

of SFB/TR 14 AVACS

Editors: Board of SFB/TR 14 AVACS

Analysis of Large Safety-Critical Systems: A quantitative Approach

by

Marc Herbstritt Ralf Wimmer

Thomas Peikenkamp Eckard Böde

Michael Adelaide Sven Johr Holger Hermanns

Bernd Becker

AVACS Technical Report No. 8

February 2006

ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog,
Andreas Podelski, Reinhard Wilhelm
ATRs (AVACS Technical Reports) are freely downloadable from `www.avacs.org`

Author(s) contact: Marc Herbstritt (`herbstri@informatik.uni-freiburg.de`).

Analysis of Large Safety-Critical Systems: A quantitative Approach*

Marc Herbstritt[1], Ralf Wimmer[1], Thomas Peikenkamp[2], Eckard Böde[2], Michael Adelaide[3], Sven Johr[4], Holger Hermanns[4], and Bernd Becker[1]

[1] Albert-Ludwigs-University, Freiburg im Breisgau, Germany
{herbstri|wimmer|becker}@informatik.uni-freiburg.de
[2] Kuratorium OFFIS e.V., Oldenburg, Germany
{thomas.peikenkamp|eckard.boede}@offis.de
[3] Carl von Ossietzky University, Oldenburg, Germany
michael.adelaide@informatik.uni-oldenburg.de
[4] Saarland University, Saarbrücken, Germany
{johr|hermanns}@cs.uni-sb.de

Abstract. The ever increasing complexity of systems requires combined efforts with respect to analysis and verification to close the so-called verification gap. On the modeling side, to face this problem, expressive high-level methodologies are used to manage system complexity. From the analysis side it is therefore essential to also start at this level. Due to powerful symbolic tools – often based on BDDs – consistent high-level *representations* can be generated. The bottleneck for subsequent system *analysis*, however, is still the incredible state space of such representations. This fact gains even more importance when the intended analysis is bound to *explicit* tools, e.g., for a quantitative analysis using stochastic model checking.

In this work, we are bridging the gaps between high-level system descriptions of safety-critical systems and corresponding explicit state space representations that can be handled by explicit quantitative analysis tools. In a first step, our approach safely integrates failure behavior of safety-critical systems into their high-level description. Then, structural reductions are applied. Manageable explicit representations are finally obtained by a novel BDD-based symbolic branching bisimulation algorithm.

We provide experimental data demonstrating the efficiency of our methods: safety-critical systems whose size seems to be prohibitive for any explicit analysis tool at the beginning are reduced by orders of magnitude, thus paving the way to quantitative analysis without losing relevant information.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

1 Introduction

The verification gap, i.e. the discrepancy between the modeling power and the capability to analyze these models, is still widening (see e.g. [1, 2]). High-level methodologies like STATEMATE [3] allow the design of very complex systems, while verification tools can in general not keep pace with this trend. But especially for safety-critical systems, where faulty system behavior can lead at best to financial losses but in the worst case has impact on humans, it is indispensable to prevent malfunctioning by rigorous system analysis. This analysis is getting even more difficult when stochastic events must be taken into account, i.e., when the occurrence of failures exhibits timed stochastic behavior.
The challenge for tackling these problems is not only to define interfaces for the corresponding modeling and analysis tools, but to develop a clear semantic flow that on the one hand enables the removal and reconstruction of parts of the system consistently so that the system is manageable, and on the other hand preserves all important system characteristics that are mandatory to check the intended properties. A solution to this problem must be able to drastically reduce the size of the system's state space, since current quantitative analysis tools are restricted to explicit domains.
Hence, the contribution of this work is a concept focusing on how statechart-based system designs (in our case STATEMATE descriptions) can be prepared and reduced in a semantically consistent way such that a quantitative analysis can be performed afterwards. A rough overview of our approach, that consists of a *discrete* and a *stochastic* part, is depicted in Figure 1.
In the *discrete* part, we are starting with a STATEMATE design and corresponding safety-relevant data (safety requirements, system failure modes, and consequential safety-critical states) an intermediate representation is derived that incorporates failure injection into the original design. Since from a requirement point of view the resulting model contains a lot of redundancies, a failure-driven cone-of-influence (COI) reduction is performed. This turns out to be an essential step to generate manageable BDD representations for the resulting labelled transition system (LTS). The final step towards an explicit model is the application of our novel *symbolic* branching bisimulation algorithm. Although bisimulation minimization techniques are known to be ineffective for standard model checking problems such as checking of invariant properties [4], in our context, we see it as the central step in compressing LTS models generated from STATEMATE to the *kernel* relevant for quantitative analysis: The branching equivalence quotient contains precisely the required information.
The *stochastic* part is not the main topic of this paper, but will be described briefly in Section 3. In principal, an interactive Markov chain (IMC) is generated from the branching bisimulation quotient. The stochastic distributions of the failures are then integrated using a compositional approach, resulting in a continous time Markov decision process (CTMPD). For this CTDMP a quantitative analysis can be performed using stochastic model checking techniques. Please note that the stochastic part is future work and is not described in detail in this paper but will be published in upcoming publications.

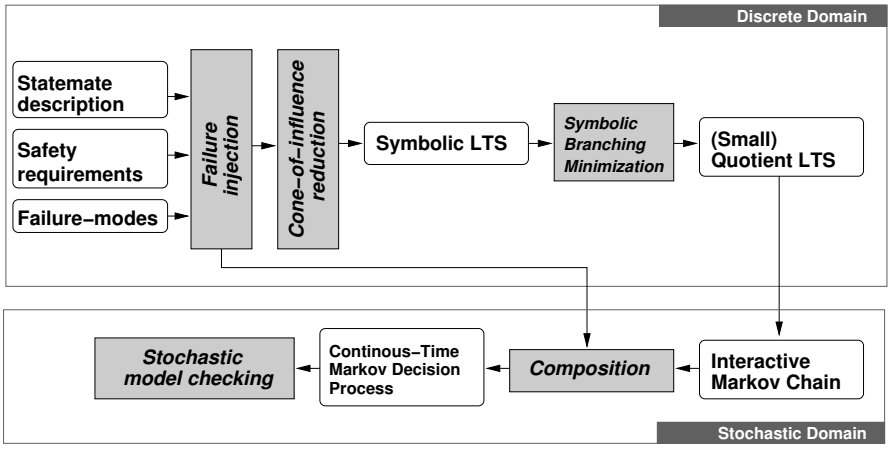

Fig. 1. Proposed tool flow for quantitative analysis of complex safety-critical systems.

The paper is structured as follows. Related work is discussed in Section 2. In Section 3 we describe the complete concept for a high-level quantitative analysis and the partial contribution of this paper to this concept. In Section 4 the compilation of STATEMATE descriptions together with failure ingredients into an LTS is described. Then, Section 5 covers our symbolic branching bisimulation. Our approach is extensively evaluated in Section 6. Finally, the paper concludes with Section 7. Details about the examples used for the evaluation of our approach are described in the appendix.

2 Related Work

With the advent of statechart-based languages like STATEMATE [3] it became possible to model large complex systems, preserving structural and functional views of the intended design under clear semantics. For non-stochastic domains there already exist verification frameworks, e.g. [5]. On the modeling side they have been extended to include failure modes attached to the model, and also on the analysis side parameterized verification algorithms have been developed to identify failure causes and consequences to allow, e.g., an automatic derivation of fault trees [6–9]. However, integrating stochastic events in a semantically sound manner is still a challenge. Several approaches exist, see [10] for an overview. Nevertheless, a link to affective stochastic verification tools has not been established so far.

One of our core steps is the application of branching bisimulation which was introduced in [11]. The basic algorithm was developed by Groote and Vandraager in [12]. Although there exists a lot of work regarding symbolic *strong* bisimulation, e.g., [13, 14], mostly focusing on process algebras, no literature can be found on BDD-based implementations for *branching* bisimulation. The work of Bouali et al. [13] mentions very shortly an extension of their approach

to branching bisimulation, but no details or even experiments are presented. A novel approach using a signature-based fixpoint algorithm was presented by Blom and Orzan in [15] for distributed computation of a quotient given an *explicit* state space representation. We analytically compared the two approaches of Groote/Vandraager and Blom/Orzan and found that the latter can be used as a basis for a symbolic, i.e., BDD-based, implementation. First results have shown that the symbolic approach is able to overcome limitations of explicit algorithms [16]. We extend these results and demonstrate that a symbolic approach adapted to the specific problem structure can be applied very successfully.
Research for quantitative system analysis has seen tremendous progress in the last decade resulting in publicly available tools, e.g., Etmcc [17], Prism [18], BcgMin, [19], and Caspa [20]. But all of them are affectively bound to *explicit* state space representations. Even Prism that incorporates clever *symbolic* data structures is finally limited to explicit domains, since the underlying numerical analysis must store explicit state informations [21]. Hence, the advantages of symbolic data structures are foiled. Accepting the fact, that quantitative analysis tools are bound to explicit state space representations, leads to the understanding that *symbolic* branching bisimulation is a mandatory step towards quantitative analysis of large systems.

3 Context

We are aiming at enabling quantitative failure analysis of real, industrial-size verification models. The cornerstones of the overall approach we pursue are as follows: (1) High-level models are generated from Statemate system descriptions. (2) After property preserving aggregations the resulting models are (3) enriched with stochastic timed quantities representing external delays – in particular, but not exclusively, component time-to-failure distributions. They are (4) analyzed via stochastic model checking with tools such as Etmcc [22] or Prism [18]. In its full generality this approach generalizes both ordinary model checking-based verification and quantitative system analysis with fault trees and similar techniques practiced in industry in one modelling framework. First experiments with this approach have shown principal feasibility [23, 19], but have also identified the main bottleneck in this 4-step modeling and analysis trajectory: Prior to stochastic model analysis, aggressive aggregation steps are required, which compress the model under consideration to the bare minimum of details needed for the stochastic analysis (see step (2) above). The reason is that state-of-the-art stochastic model checking relies on approximate solution algorithms for systems of linear equations or inequations [24–26]. These algorithms, even if implemented in a symbolic or semi-symbolic setting [21] are restricted to model sizes in the range of 10^5 to 10^6 states.
To provide such aggressive aggregations we are applying the concept of branching bisimilarity. Although it has been reported that (strong) bisimulation minimization is ineffective for standard model checking problems such as checking of invariant properties [4], we see it as a central step in compressing LTS models

generated from STATEMATE, since the branching equivalence quotient contains precisely the information required in our approach. This is due to the fact that branching bisimulation preserves CTL without Next-state operator [27]. As evident from the experiments in Section 6, building the branching bisimulation quotient with our novel algorithm allows us to accomplish the above step (2): we manage to compress the model to the required range, thereby preserving all interesting model properties. In particular nested timed reachability properties like "The probability of reaching a state within 2 minutes from where it is always possible to survive a catastrophic failure for at least 20 minutes is greater than p" are preserved. These properties, known as *survivability*, are of prime interest for safety-critical systems.
The overarching model for our tool chain is the model of interactive Markov chains (IMC), which contains ordinary LTS and ordinary Markov chains (in continuous time) as fragments. Therefore, building the quotient of an LTS with respect to branching equivalence is assured to preserve properties of an appropriate CTL-like logic on IMC, that is, a conservative integration of CSL-X [25] on Markov chains, and PCTL-X on Markov decision processes [24]. In particular, nested timed reachability properties like "The probability of reaching a state within 2 minutes from where it is always possible to survive a catastrophic failure for at least 20 minutes is greater than p" are preserved.
Finally, the IMC and the failure distributions are combined in a compositional way (see [23]) and results in a continous-time Markov decision process for which stochastic model checking can be applied, e.g., to check survivability properties. More details on how to model check such properties can be found in [19].

4 Model Construction

In this section we describe the compilation of STATEMATE designs together with a specification of failure occurences into a labelled transition system (LTS). Additionally, we describe reduction mechanisms to get manageable BDD representations of the corresponding LTS.

4.1 Compilation

The upper left part of Figure 1 shows the principal translation and analysis steps necessary to successfully perform the reduction. The main ingredients of the problem specification are, first, a STATEMATE model specifying the nominal behaviour of the system, second, a number of failures, that are injected, leading to a model encompassing also the disfunctional behaviour and, third, a specification of safety critical states. The rationale for keeping the failures separate from the model is already described in [9]. From an analysis point of view it is important to keep order and branching of failures untouched in all of the following transformation steps. This is due to the need of extracting failure sequences for backtranslation in a post-analysis step, since the real system underlies some inherent timing behavior. In the first step of the compilation the

STATEMATE model is translated into a simple intermediate language by replacing graphical items of the specification language by state variables, replacing structured data types by simple ones, and discretizing continuous variables (see [28] for details). The generated intermediate model is intended to represent the nominal behaviour. It can be viewed as a LTS $M = (S, A, T)$ where S is a finite set of states, A a finite set of actions including the unobservable action τ and $T \subseteq S \times A \times S$ is a set of transitions. Please note that the τ-action will play a crucial role in our approach, since it is used for abstracting the nominal behavior of the system.

4.2 Failure injection

In a second step we are injecting failure behavior in our model by an instantiation of elements of a failure library. The behaviour associated with these elements is parameterized so that, e.g., *delay failures* have a parameter indicating the (maximal) delay introduced with the particular instance of the failure. The concrete values of these parameters are specified during instantiation. Please note that we don't actually specify the behaviour of the injected failures here, since it is sufficient to know that the system extended with a set of failures F "includes" the original system in the following sense: A labeled transition system $M' = (S, A', T')$ is called an F-extension of $M = (S, A, T)$ iff $A' = A \dot{\cup} F$, $T' \supseteq T$, and $\forall s \in S. \forall f \in F. \exists s' \in S : (s, f, s') \in T'$. It should be noted that failures can occur at any time according to this definition, the rationale being that the occurence of failures has not been taken into account during model construction and therefore we may not exclude some occurences of failures *a priori*.

4.3 Safety-critical states

After introducing the failure behavior, we are now able to identify safety-critical states, i.e., states where catastrophic failures occur according to the safety requirements. This can be used as a problem-specific initial partition of the state space into *non-critical* and *critical* states, and will be used as a starting point for our branching bisimulation algorithm described in Section 5.

4.4 Cone-of-influence reduction and introduction of τ-actions

The specification of safety-critical states can also be used for strong reductions on the size of the model by performing a cone-of-influence reduction (COI) w.r.t. these states [29]. Whereas on the symbolic representation of the model the reduction is achieved by eliminating variables, on the transition system the reduction amounts to collapsing a huge number of paths. Being a standard technique in model checking, this step may look somewhat unusual in the context of the quotient construction outlined in Section 5. However, since we are only interested in the contribution of failures to (the reachability of) safety-critical states, this step is consistent. Our experiments show that COI reduction

opens the gates for handling large designs and additionally is leading to small models. A further reduction on the symbolic representation is obtained by replacing all non-failure actions, i.e., labels not in F, by the unique τ action. The resulting transition system together with the initial partition is then passed to the symbolic branching bisimulation algorithm being described in the next section.

4.5 BDD generation of LTS

After the reductions steps the reduced model is translated into a model with binary values only. The result is a BLIF-MV file suitable for the VIS model checker [30]. It is used to flatten the hierachy of the design and to transform the net-list-like input format into a relational presentation of the transition relation. The generation of the monolithic transition relation, that must be provided to the symbolic branching bisimulation algorithm, is improved by using a partitioned transition relation [31] for optimization techniques like variable quantification, since many of these BDDs depend only on some of the model variables. All this steps are performed using VIS. At the end we have a symbolic BDD-representation of our LTS.

5 Symbolic Branching Bisimulation

In this section we describe the implementation of our branching bisimulation algorithm that works uniformly with BDDs. We assume that the reader is familiar with standard BDDs and the corresponding algorithms. For a comprehensive treatment see e.g. [32].

5.1 Basic algorithm

To shorten writing we introduce some notations. For a labeled transition system $M = (S, A, T)$ and a partition P of S we write: $s \xrightarrow{a} t$ for $(s, a, t) \in T$, $\xrightarrow{a^*}$ for the reflexive transitive closure of $\xrightarrow{a}$, $\xrightarrow[P]{a}$ for a transition that is inert w.r.t. P, and $\xrightarrow[P]{a^*}$ for the reflexive transitive closure of $\xrightarrow[P]{a}$. Inert means that the source and target state of a transition are contained in the same block. By $P(s)$ we denote the block of P that contains the state s, i.e. $P(s) = \{t \in S \mid \exists B \in P : s \in B \wedge t \in B\}$. A branching bisimulation according to [33] is defined as follows.

Definition 1. *Given a LTS $M = (S, A, T)$. Then, a relation $R \subseteq S \times S$ is a* branching bisimulation *if R is symmetric and for all $s_1, s_2, t_1 \in S$ the following holds: If $(s_1, t_1) \in R$ and $s_1 \xrightarrow{a} s_2$ then*

$$\begin{aligned} &\textit{either} \quad a = \tau \textit{ and } (s_2, t_1) \in R \\ &\textit{or} \quad \exists t_1', t_2 \in S : t_1 \xrightarrow{\tau^*} t_1' \xrightarrow{a} t_2 \wedge (s_1, t_1') \in R \wedge (s_2, t_2) \in R \end{aligned}$$

Algorithm 1 Computation of the coarsest branching bisimulation

```
1: procedure BranchingBisimulation
2:     P_old ← ∅, P ← initial partition
3:     while P_old ≠ P do
4:         P_old ← P
5:         for all blocks B of P_old do
6:             P ← P \ {B} ∪ sigref(B)
7:         end for
8:     end while
9:     return P
10: end procedure
```

In [15], Blom and Orzan presented a novel approach for the distributed computation of branching bisimulation. Their algorithm is based on analyzing the *signatures* of states w.r.t. to the current partition. The signature of a state is like a fingerprint identifying possible actions that can be executed in this state. To preserve branching bisimilarity, the unobservable action τ must be taken into account by ignoring sequences of τ-actions that never leave the corresponding partition block of the state. Then, a refinement of a partition can be computed by dividing blocks by identifying states that have the same signature. Formally, this is captured in the following definition.

Definition 2. *Let $P = \{B_0, \ldots, B_{p-1}\}$ be a partition of the state space S. The signature* $\mathrm{sig}_P(s)$ *of a state s regarding the partition P is defined as*

$$\mathrm{sig}_P(s) = \{(a, B_i) \mid \exists s', s'' \in S : s \xrightarrow[P]{\tau^*} s' \xrightarrow{a} s'' \in B_i \wedge (a \neq \tau \vee s \notin B_i)\}.$$

The refinement $\mathrm{sigref}(P)$ *of a partition P regarding the signatures of the states is defined as*

$$\mathrm{sigref}(P) = \left\{\{s' \in S \mid \mathrm{sig}_P(s) = \mathrm{sig}_P(s')\} \,\middle|\, s \in S\right\}.$$

The fixpoint algorithm of [15] is sketched in Algorithm 1. But in contrast to the original algorithm of Blom and Orzan, we had to modify the algorithm such that we can start with an initial partition, as already mentioned in Section 4, for separating critical and non-critical states. This is not a restriction, but a useful extension of the algorithm, since we are now able to initially partition the state space according to the analysis problem. Additionally, we have integrated a simple, but efficient optimization technique that was not applied in [15]. The idea is to handle not all blocks together, but to take one block at a time. For this block the signature refinement is computed and the corresponding result is updated *in situ* in the current partition. This *block forwarding* technique results in impressive speedups within our experiments due to the reduced number of iterations of the fixpoint algorithm (see section 6.1 for details).
Now we will turn to symbolic representations of LTSs and how we can compute the branching bisimulation quotient on the BDD-level.

Algorithm 2 Computation of the Signatures

1: **procedure** SIGNATURES(states σ, transitions $\mathcal{T}$, partition $\mathcal{P}$)
2: $\quad \mathcal{T}_{\tau,inert}(s,t) = (\sigma(s) \wedge \sigma(t) \wedge \mathcal{T}(s,a,t))\ .Cofactor(a = \tau)$
3: $\quad reltrans(s,a,t) \leftarrow \sigma(s) \wedge \mathcal{T}(s,a,t) \wedge \neg\mathcal{T}_{\tau,inert}(s,t)$
4: $\quad targets(s,a,k) \leftarrow \exists t : (reltrans(s,a,t) \wedge \mathcal{P}(t,k))$
5: $\quad C_\tau(s,t) \leftarrow computeClosure(\mathcal{T}_{\tau,\text{inert}})$
6: $\quad \mathcal{S}(s,a,k) \leftarrow \exists t : (C_\tau(s,t) \wedge targets(t,a,k))$
7: $\quad$ **return** $\mathcal{S}$
8: **end procedure**

5.2 BDD representations

We have to represent the state space S, the transition relation T, the partition P, and the signatures *sig* as BDDs. Regarding the BDD for the partition we assign a unique number to each block and use a binary encoding for the states[5], the actions (variables a) and the block numbers (variables k). Then, the state space is encoded by a BDD σ such that $\sigma(s) = 1$ iff $s \in S$. Analogically, the transitions relation is represented as a BDD $\mathcal{T}$ with $\mathcal{T}(s,a,t) = 1$ iff $s \xrightarrow{a} t$. For the partition we have a BDD $\mathcal{P}$ with $\mathcal{P}(s,k) = 1$ iff $s \in B_k$, and for the signatures we get a BDD $\mathcal{S}$ with $\mathcal{S}(s,a,k) = 1$ iff $(a, B_k) \in \text{sig}(s)$.

Signature Computation. The algorithm for the symbolic computation of the signatures for a set of states regarding a partition P is shown in Algorithm 2. In lines 2 and 3 the transition relation is split into inert τ-transitions ($\mathcal{T}_{\tau,\text{inert}}$) and the remaining ones (*reltrans*). Thus, *reltrans* contains all transitions that satisfy the condition $a \neq \tau$ or connect two different blocks of the current partition. Next, the target state of the transition is substituted by its block number (line 4). All those states must be taken into account that have an outgoing relevant transition by arbitrarily many inert τ-transitions. For this, in line 5, the reflexive transitive closure of inert τ-transitions must be computed (e.g., using [34]). Finally, the states of the closure are added to the signatures.

5.3 Refinement

Given the BDD for the signatures of all states, we will now compute a new partition where all states with the same signature are merged into one block. To do so, we are placing the s_i-variables at the beginning of the variable order of the BDDs. Then, $\text{level}(s_i) < \text{level}(a_j)$ and $\text{level}(s_i) < \text{level}(k_l)$ hold for all i, j and l. This enables us to exploit the following observation. Let s be the encoding of a state and v the BDD node that is reached when following the path from the BDD root according to s. Then, the sub-BDD at v is a representation of the signature of s. The point is, that for all states having the same signature as s the corresponding paths lead also to this BDD-node v. Therefore, to get the

[5] Variables s, t, and x are used as current state, next state, and auxiliary variables.

Algorithm 3 Computation of the Refined Partition

```
1: procedure REFINE(signatures S)
2:     if S ∈ ComputedTable then return ComputedTable[S]
3:     end if
4:     x ← topVar(S)
5:     if x = s_i then
6:         low ← Refine(S_x̄)
7:         high ← Refine(S_x)
8:         result ← ITE(x, high, low)
9:     else
10:        result ← newBlockNumber()
11:    end if
12:    ComputedTable[S] ← result;
13:    return result
14: end procedure
```

new block that contains s and all other states having the same signature as s, we simply have to replace the sub-BDD at v by the BDD for the encoding of the new block number k. Algorithm 3 sketches the description above. It makes use of a function `newBlockNumber` that returns the BDD-encoding of a number that has not been already used within this iteration. The corresponding internal counter is reset each time we call `Refine`. When the node is not labeled with a state variable s_i, the sub-BDD at this node represents a signature that we have to substitute with a new block number. Otherwise, when the node is labeled with a state variable we call the algorithm recursively for the two sons of that node. Furthermore, the algorithm relies on a dynamic programming approach using a so-called *ComputedTable* to prevent redundant computations.

5.4 Quotient LTS extraction

In the end, we only have to extract the quotient LTS from the final partition. For this, let $\mathcal{P}$ be a partition – represented as described above as a BDD – with sigref($\mathcal{P}$) = $\mathcal{P}$. To extract the quotient systems regarding this partition, we have to collapse the states of each block into one new state whose encoding is the same as the block number:

$$\sigma_{new}(s) := [k \to s](\exists s : \mathcal{P}(s,k))$$

The new transition relation can be computed as follows:

$$\mathcal{R}(s,a,t) := [k \to t](\exists t : \mathcal{T}(s,a,t) \wedge \mathcal{P}(t,k))$$
$$\mathcal{T}_{new}(s,a,t) := [k \to s](\exists s : \mathcal{R}(s,a,t) \wedge \mathcal{P}(s,k))$$

The notation $[k \to t]$ means that the k-variables must be renamed to the corresponding t-variables.

6 Experimental Results

In this section we present and discuss our experimental results that we have performed first for process algebraic system descriptions as a pre-evaluation, and secondly for safety-critical STATEMATE designs.

6.1 Evaluation of Symbolic Branching Bisimulation

To check the efficiency of our symbolic branching bisimulation we performed preliminary experiments on a process algebraic description for a Kanban system [35] that models a production environment having buffers for a parameterizable number of workpieces at each machine. To generate symbolic BDD-representations we have applied CASPA [20]. We performed two series of experiments with two different configurations of the Kanban system. In the first configuration we have hidden all internal process actions, such that only the synchronization actions were visible. This kind of configuration could be of interest when inter-process communication only is analyzed. In the second configuration we have hidden all actions but the actions that are related to the first of the four processes. The motivation for this configuration is that one only wants to analyze the first process and likes to ignore the others. More details about the Kanban example can be found in Appendix A.
Since we are not aware of any existing *symbolic* branching bisimulation tool, we compared our symbolic branching bisimulation tool SIGREF against BCGMIN [19] from the CADP toolbox [36]. BCGMIN provides the explicit Groote-Vaandrager algorithm. Since BCGMIN requires as input an explicit description of the transition system, we had to generate a file containing all transitions explicitly (in the `.bcg` format).
Tables 1 and 2 show the results for each configuration.[6] In both tables, column 1 denotes the number of workpieces along which the Kanban description can be parameterized. The second column gives the original number of states encoded in the LTS generated from CASPA. As it is shown in column 3, the state space contains also unreachable states. [7] Although the number of states is already impressive, the corresponding number of transitions is even always one order of magnitude bigger. The size of the bisimulation quotient is given in the last two columns. It becomes clear that these models exhibit a large amount of compression potential regarding branching bisimilarity. In the middle of Table 1 the performance values of BCGMIN and SIGREF are given in terms of memory consumption and CPU runtime. For instances having more than 20 million states, BCGMIN fails although the bisimulation quotient itself is very small. Clearly, the reason is that BCGMIN cannot handle the original LTS explicitly.

[6] The experiments were performed on an AMD Dual Opteron Processor with 2.6GHz and a main memory limit of 3 GB under Debian Linux.

[7] Unreachable states are a typical artifact immanent to symbolic encodings, but inconvenient in explicit domains.

p	states		transitions		SIGREF			BCGMIN		bisim. quotient	
	all	reachable	all	reachable	BDD peak	memory	time	memory	time	states	transitions
1	256	160	904	616	16352	9 MB	0.02s	3 MB	0.19s	24	42
2	63808	4600	231424	28120	435372	23 MB	0.58s	8 MB	0.47s	206	552
3	$1.02 \cdot 10^6$	58400	$4.65 \cdot 10^6$	446400	2212630	59 MB	11.20s	92 MB	8.73s	872	2968
4	$1.60 \cdot 10^7$	454475	$7.44 \cdot 10^7$	$3.98 \cdot 10^6$	2313808	63 MB	1m 38s	1457 MB	2m 46s	2785	10932
5	$1.68 \cdot 10^7$	$2.54 \cdot 10^6$	$1.33 \cdot 10^8$	$2.44 \cdot 10^7$	2483460	67 MB	9m 15s	2365 MB	7m 48s	7366	31795
6	$2.64 \cdot 10^8$	$1.12 \cdot 10^7$	$1.68 \cdot 10^9$	$1.15 \cdot 10^8$	2704212	76 MB	40m 24s	memout	—	17010	78584
7	$2.68 \cdot 10^8$	$4.16 \cdot 10^7$	$2.61 \cdot 10^9$	$4.50 \cdot 10^8$	5037438	124 MB	4h 49m 36s	memout	—	35456	172382
8	$4.22 \cdot 10^9$	$1.33 \cdot 10^8$	$2.90 \cdot 10^{10}$	$1.51 \cdot 10^9$	11281858	244 MB	11h 59m 00s	memout	—	68217	345128

Table 1. Results for the Kanban system where process internal actions are hidden.

p	states		transitions		SIGREF			BCGMIN		bisim. quotient	
	all	reachable	all	reachable	BDD peak	memory	time	memory	time	states	transitions
1	256	40	916	117	19418	9 MB	0.02s	3 MB	0.17s	32	68
2	63808	460	234496	2136	308644	15 MB	0.40s	8 MB	0.53s	200	680
3	$1.02 \cdot 10^6$	2920	$4.72 \cdot 10^6$	16940	1897854	54 MB	4.54s	93 MB	10.46s	800	3400
4	$1.60 \cdot 10^7$	12985	$7.56 \cdot 10^7$	86220	2249422	62 MB	1m 13s	1474 MB	3m 42s	2540	11900
5	$1.68 \cdot 10^7$	45472	$1.36 \cdot 10^8$	330897	2392502	66 MB	9m 49s	2400 MB	13m 04s	6272	33320
6	$2.64 \cdot 10^8$	134064	$1.72 \cdot 10^9$	$1.04 \cdot 10^6$	2561132	74 MB	1h 28m 57s	memout	—	14112	79968
7	$2.68 \cdot 10^8$	347040	$2.65 \cdot 10^9$	$2.84 \cdot 10^6$	4752300	117 MB	9h 10m 15s	memout	—	28800	171360
8	$4.22 \cdot 10^9$	811305	$2.95 \cdot 10^{10}$	$6.90 \cdot 10^6$	7823410	181 MB	46h 59m 33s	memout	—	54450	336600

Table 2. Results for the Kanban system where all but the actions related to the first process are hidden.

p	without block forwarding # iterations	time [s]	with block forwarding # iterations	time [s]	factor
1	5	0.02	4	0.02	1.00
2	8	0.57	6	0.58	1.00
3	11	17.14	7	11.20	1.53
4	14	181.33	8	98.02	1.85
5	17	1243.29	8	555.04	2.24
6	20	6811.16	8	2423.90	2.81
7	23	34925.16	10	17375.70	2.01
8	26	137186.60	10	43140.44	3.18

Table 3. Comparison of SIGREF with and without block forwarding for the first Kanban configuration where all but process internal actions are hidden.

But it is interesting to see that our symbolic tool SIGREF handles instances of up to 4 billion states and nearly 30 billion transitions. In particular the memory consumption of SIGREF is very robust: for the largest instance we have a BDD peak node of about 11 million, and for all others it is less than 8 million.
Tables 3 and 4 give details about the performance of our block forwarding optimization as described in Section 5.1. In columns 2 and 4 the number of iterations of the fixpoint algorithm for branching bisimulation are given having block forwarding disabled and enabled, respectively. Columns 3 and 5 give the corresponding overall CPU runtimes. The speedup factor is given in the last column. For the second Kanban configuration (Table 4) disabling block forwarding is always better. But for the real time consuming instances, i.e., ($p \geq 5$), enabling block forwarding is only 20-30% worse, especially for $p = 8$ we only lose 12%. The picture is different for the first Kanban configuration (Table 3). There, block forwarding always yields a tremendous speedup. For the larger instances ($p \geq 5$) the speedup factor ranges from 2.01 to 3.18. Especially for $p = 8$, SIGREF having block forwarding enabled is more than three times faster. Putting the results of Tables 3 and 4 together, it is clear that block forwarding should be enabled.

6.2 Evaluation for safety-critical STATEMATE designs

Due to the promising results from the experiments of Section 6.1, we continued to further validate our approach for STATEMATE designs that focus on typical safety-critical domains, e.g., train systems and avionics.

Benchmarks. We have developed novel models stemming from the European Train Control System (ETCS) specification: ETCS-1,ETCS-2,ETCS-3. These benchmarks model a scenario regarding the communication between trains and the Radio Block Centers (RBCs) (see [37] for details about the ETCS specification which is part of the ERTMS project). The analysis tackles the problem of colliding trains on the same track. The example is scalable regarding the number

p	without block forwarding # iterations	time [s]	with block forwarding # iterations	time [s]	factor
1	4	0.01	4	0.02	0.50
2	4	0.31	4	0.40	0.78
3	4	4.18	4	4.54	0.92
4	5	52.15	5	73.45	0.71
5	6	507.31	6	589.89	0.86
6	7	3843.31	7	5337.93	0.72
7	8	26742.47	8	33015.39	0.81
8	9	148872.57	9	169173.37	0.88

Table 4. Comparison of SIGREF with and without block forwarding for the second Kanban configuration where all actions but the actions related to the first process are hidden.

of trains whereby we used 1, 2, and 3 trains. Especially ETCS-3 samples a scenario of realistic size. From the ARP 4761 case study [38] we have taken two benchmarks (BS-S,BS-P) that model a braking system from an airplane. They are about the correct functioning between the pilot's braking pedal and the hydraulic pressure given to the wheels of the airplane. The benchmarks CTRL and CTRL-A are derived from a redundancy controller that is taken from another industrial avionics project. All, but the CTRL, benchmarks are made publicly available via our website [39] by providing the STATEMATE working areas, the intermediate BLIF-MV files, and XML descriptions of the BDDs for the original LTS and its bisimulation quotient. More details about the ETCS and the braking system example can be found in Appendix B and C.

Evaluation. In Tables 5 and 6 we have compared the generated LTSs regarding the failure-driven cone-of-influence reduction as described in Section 4.[8] Column 1 denotes the benchmark. Column 2 (3) refer to all (reachable) states, and column 4 (5) refer to all (reachable) transitions[9]. Because τ-actions play a crucial role for branching bisimulation, they are listed in columns 6 and 7, respectively. The tables indicate that typically the number of failure (i.e. non-τ) transitions strongly exceeds the number of τ-transitions. This is due to the failure injection mechanism that introduces for each nominal (i.e. τ) action a number of failure actions proportional to the number of failures. The results clearly show that COI reduction reduces the generated state spaces immense, and for the two largest instances, ETCS-3 and CTRL-A, we were not even able to generate the LTSs without COI within 48 hours CPU running time.
Finally, Table 7 contains the results for the whole proposed tool flow, including COI reduction, LTS generation, and symbolic branching bisimulation. At first, we have applied our tool SM2LTS that generates a BDD-based LTS representation out of a STATEMATE description together with the failure modes according to Section 4. SM2LTS is built upon the VIS verification framework [30]. Please note

[8] The experiments of Table 5, 6, and 7 were performed on a Sun-Fire-V490 running Solaris 5.10 with a main memory limit set to 4GB.
[9] A transition is *reachable* if it emanates from a reachable state.

Model	#a	states		transitions		τ-transitions	
		all	reachable	all	reachable	all	reachable
ETCS-1	9	$8.39 \cdot 10^6$	1056	$2.68 \cdot 10^9$	337920	$2.68 \cdot 10^8$	33792
ETCS-2	17	$2.2 \cdot 10^{12}$	428112	$4.05 \cdot 10^{16}$	$7.89 \cdot 10^9$	$2.25 \cdot 10^{15}$	$4.38 \cdot 10^8$
ETCS-3	25	—	—	—	—	—	—
BS-S	13	$4.19 \cdot 10^6$	488032	$1.88 \cdot 10^9$	$2.19 \cdot 10^8$	$1.34 \cdot 10^8$	$1.56 \cdot 10^7$
BS-P	17	$1.72 \cdot 10^{10}$	$1.85 \cdot 10^8$	$9.9 \cdot 10^{12}$	$1.06 \cdot 10^{11}$	$5.5 \cdot 10^{11}$	$5.92 \cdot 10^9$
CTRL	11	$9.01 \cdot 10^{15}$	139623	$2.16 \cdot 10^{17}$	$3.35 \cdot 10^6$	$9.01 \cdot 10^{15}$	279244
CTRL-A	22	—	—	—	—	—	—

Table 5. LTS generation without cone-of-influence reduction.

Model	#a	states		transitions		τ-transitions	
		all	reachable	all	reachable	all	reachable
ETCS-1	9	$4.19 \cdot 10^6$	1056	$1.34 \cdot 10^9$	337920	$1.34 \cdot 10^{08}$	33792
ETCS-2	17	$5.5 \cdot 10^{11}$	428112	$1.01 \cdot 10^{16}$	$7.89 \cdot 10^9$	$5.63 \cdot 10^{14}$	$4.38 \cdot 10^{08}$
ETCS-3	25	$7.21 \cdot 10^{16}$	$1.59 \cdot 10^8$	$6.14 \cdot 10^{22}$	$1.35 \cdot 10^{14}$	$2.36 \cdot 10^{21}$	$5.2 \cdot 10^{12}$
BS-S	8	512	320	9216	5760	1024	640
BS-P	17	$1.72 \cdot 10^{10}$	$1.85 \cdot 10^8$	$6.18 \cdot 10^{11}$	$6.66 \cdot 10^9$	$3.44 \cdot 10^{10}$	$3.7 \cdot 10^8$
CTRL	11	$2.75 \cdot 10^{11}$	139622	$2.64 \cdot 10^{13}$	$1.34 \cdot 10^7$	$2.2 \cdot 10^{12}$	$1.12 \cdot 10^6$
CTRL-A	22	$2.75 \cdot 10^{11}$	$1.51 \cdot 10^6$	$5.06 \cdot 10^{13}$	$2.78 \cdot 10^8$	$2.2 \cdot 10^{12}$	$1.21 \cdot 10^7$

Table 6. LTS generation with cone-of-influence reduction.

that our proposed COI reduction is also performed by Sm2Lts. Secondly, we applied Sigref for further minimization. Therefore, the two three-fold columns in Table 7 give details about the performance of Sm2Lts and Sigref, respectively. For both tools we recorded the peak number of BDD nodes, upper bounds for memory consumption, and CPU running times.[10] Again, the last two columns show the size of the corresponding bisimulation quotient.

The overall results for the size of the branching bisimulation quotient show that we are able to minimize the systems up to a factor of $2 \cdot 10^{12}$ for ETCS-3. Most important is, however, that the resulting state spaces for all examples lie in the range that can be handled by explicit tools.

Sm2Lts has a very good performance and is finishing in all cases within some minutes. The memory requirements of Sm2Lts generally seem to be harmless, although for CTRL-A we had to spend 2GB. But the results show that the computational complexity lies in computing the branching bisimulation.

However, Sigref terminates for all but CTRL-A with a reasonable memory consumption. The time required by Sigref ranges from seconds to several hours. Unfortunately, we fail on the CTRL-A example, but this also shows that our examples are definitely non-trivial. But even the long runtimes for the large ETCS-3 example are acceptable, if we recall the tremendous state space reduction. Especially, ETCS-2, ETCS-3, BS-P, and CTRL are far out of scope for explicit bisimulation tools.

[10] These numbers refer to the translation from BLIF-MV to LTS using VIS.

Model	SM2LTS			SIGREF			bisim. quotient	
	BDD peak	memory	time	BDD peak	memory	time	states	transitions
ETCS-1	15330	<64 MB	0.8s	181916	<64 MB	0.5s	50	739
ETCS-2	183960	<64 MB	13.2s	$2.22 \cdot 10^7$	<512 MB	4m 37s	1311	48830
ETCS-3	566188	<128 MB	22.2s	$1.15 \cdot 10^8$	<2 GB	14h 7m 39s	35841	$3.13 \cdot 10^6$
BS-S	3066	<64 MB	0.2s	22484	<64 MB	0.1s	6	29
BS-P	134904	<64 MB	8.4s	$7.64 \cdot 10^7$	<2 GB	47m 29s	1176	42812
CTRL	$1.33 \cdot 10^6$	<128 MB	6m 11s	$2.73 \cdot 10^7$	<1 GB	1h 21m 57s	9626	653291
CTRL-A	$1.27 \cdot 10^7$	<2 GB	5m 40s	—	memout	—	—	—

Table 7. Results for LTS generation from STATEMATE using cone-of-influence reduction and subsequent branching minimization.

As a conclusion, the results are very promising and clearly show the feasibility of our approach.

7 Conclusions

In this work we have presented a method to "shrink-fit" extremely large safety-critical systems to a size that makes them feasible for quantitative analysis. Our approach is based on an efficient compilation of STATEMATE designs and corresponding failures into a BDD-based LTS representation which is further processed using symbolic branching bisimulation. We have performed various experiments showing that we are able to reduce systems of several billion states to small models, thereby preserving relevant system characteristics which are mandatory for a subsequent quantitative analysis.
Future work will further concentrate on implementing the remaining stochastic parts to complete the big picture.

Acknowledgements

This research was to a great extent only made possible by inspiring discussions with our colleagues, especially Erika Ábrahám. Additionally we would like to thank Hubert Garavel for providing us insight into his most interesting work. Furthermore, we are grateful to Markus Siegle and Matthias Kuntz for the supply of their CASPA tool. Last but not least, we'd like to thank Matthias Pretzer for making available the braking system example, and Samuel Wischmeyer for documenting the benchmarks.

References

1. Kunz, W.: State-of-the-art in Equivalence Checking (2003) Invited tutorial, Seminar on Formal Verification Methods (Roeros, Norway).
2. Smith, S.: 2005: The Year of Verification Reuse (Jan 27 2005) online at: `http://www.techonline.com/community/ed_resource/feature_article/37575`.
3. Harel, D., Politi, M.: Modelling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw-Hill (1998)

4. Fisler, K., Vardi, M.Y.: Bisimulation and model checking. In Pierre, L., Kropf, T., eds.: Proc. of CHARME. Vol. 1703 of LNCS. (1999) 338–341
5. Bienmüller, T., Damm, W., Wittke, H.: The STATEMATE Verification Environment - Making It Real. In: Proc. of CAV. Vol. 1855 of LNCS. (2000) 561–567
6. Bozzano, M., Villafiorita, A., Åkerlund, O., Bieber, P., Bougnol, C., Böde, E., Bretschneider, M., Cavallo, A., Castel, C., Cifaldi, M., Cimatti, A., Griffault, A., Kehren, C., Lawrence, B., Lüdtke, A., Metge, S., Papadopoulos, C., Passarello, R., Peikenkamp, T., Persson, P., Seguin, C., Trotta, L., Valacca, L., Zacco, G.: ESACS: An integrated methodology for design and safety analysis of complex systems. ESREL (2003)
7. Åkerlund, O., Engblom, J., Werner, B., Bieber, P., Castel, C., L.Sagaspe, Seguin, C., Böde, E., Lüdke, A., Peikenkamp, T., Bolzano, M., Bretschneider, M., Cruz, M.F.D., Frisk, M., Metge, S., Papadopoulos, C., Trivedi, H., Cavallo, A., Cifaldi, M., Gauthier, J., Griffault, A., Lisagor, O., Person, P.: ISAAC, a framework for integrated safety analyses of functional, geometrical and human aspects. ERTS (2006)
8. Vesely, W.E., Goldberg, F., Roberts, N.H., Haasl, D.F.: Fault Tree Handbook. NUREG-0492. U.S. Nuclear Regulatory Commission, Washington DC (1981)
9. Peikenkamp, T., Böde, E., Brückner, I., Spenke, H., Bretschneider, M., Holberg, H.: Model-based Safety Analysis of a Flap Control System. In: Proc. of INCOSE, Toulouse (2004)
10. Jansen, D.N., Hermanns, H.: QoS modelling and analysis with UML-statecharts: the StoCharts approach. SIGMETRICS Performance Evaluation Review **32**(4) (2005) 28–33
11. Browne, M., Clarke, E., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theoretical Computer Science **59** (1988) 115–131
12. Groote, J.F., Vaandrager, F.W.: An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In Paterson, M.S., ed.: Automata, Languages and Programming. Vol. 443 of LNCS. (1990) 626–638
13. Bouali, A., de Simone, R.: Symbolic bisimulation minimisation. In: Proc. of CAV. Vol. 663 of LNCS. (1992) 96–108
14. Enders, R., Filkorn, T., Taubner, D.: Generating BDDs for Symbolic Model Checking in CCS. Distributed Computing **6**(3) (1993) 155–164
15. Blom, S., Orzan, S.: Distributed branching bisimulation reduction of state spaces. In: Proc. of Int'l Work. on Parallel and Distributed Model Checking. (2003)
16. Wimmer, R., Herbstritt, M., Becker, B.: Minimization of Large State Spaces using Symbolic Branching Bisimulation. submitted to IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS) (2006)
17. Hermanns, H., Katoen, J.P., Meyer-Kayser, J., Siegle, M.: A markov chain model checker. In: Proc. of TACAS. Vol. 1785 of LNCS. (2000) 347–362
18. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. J. STTT **6**(2) (2004) 128–142
19. Garavel, H., Hermanns, H.: On Combining Functional Verification and Performance Evaluation Using CADP. In: Prof. of FME. Vol. 2391 of LNCS. (2002)
20. Kuntz, M., Siegle, M., Werner, E.: Symbolic performance and dependability evaluation with the tool CASPA. In: Proc. of FORTE. Vol. 3236 of LNCS. (2004) 293–307
21. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems. PhD thesis, University of Birmingham (2002)
22. Hermanns, H., Katoen, J.P., Meyer-Kayser, J., Siegle, M.: A Tool for Model-Checking Markov Chains. J. STTT **4**(2) (2003) 153–172
23. Hermanns, H., Katoen, J.P.: Automated compositional Markov chain generation for a plain-old telephone system. Science of Comp. Programming **36** (2000) 97–127

24. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In Thiagarajan, P.S., ed.: Proc. of FSTTCS. Vol. 1026 of LNCS. (1995) 499–513
25. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time markov chains. IEEE Trans. Software Eng. **29**(6) (2003) 524–541
26. Baier, C., Hermanns, H., Katoen, J.P., Haverkort, B.R.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time markov decision processes. Theor. Comput. Sci. **345**(1) (2005) 2–26
27. Nicola, R.D., Vaandrager, F.: Three logics for branching bisimulation. J. ACM **42**(2) (1995) 458–487
28. Bienmüller, T., Brockmeyer, U., Damm, W., Döhmen, G., Eßmann, C., Holberg, H.J., Hungar, H., Josko, B., Schlör, R., Wittich, G., Wittke, H., Clements, G., Rowlands, J., Sefton, E.: Formal Verification of an Avionics Application using Abstraction and Symbolic Model Checking. In Redmill, F., Anderson, T., eds.: Proc. of Safety-critical Systems Symposium, Safety-Critical Systems Club (1999) 150–173
29. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
30. The VIS Group: VIS: A system for verification and synthesis. In Alur, R., Henzinger, T., eds.: Proc. of CAV. Vol. 1102 of LNCS. (1996)
31. Ranjan, R., Aziz, A., Brayton, R., Plessier, B., Pixley, C.: Efficient BDD algorithms for FSM synthesis and verification. In: Proc. of IWLS. (1995)
32. Drechsler, R., Becker, B.: Binary Decision Diagrams – Theory and Implementation. Kluwer Academic Publishers (1998)
33. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. J. ACM **43**(3) (1996) 555–600
34. Matsunaga, Y., McGeer, P.C., Brayton, R.K.: On Computing the Transitive Closure of a State Transition Relation. In: Proc. of DAC. (1993) 260–265
35. Ciardo, G., Tilgner, M.: On the use of Kronecker operators for the solution of generalized stochastic Petri nets. Technical Report 96-35, Institute for Computer Applications in Science and Engineering (1996)
36. Garavel, H., Lang, F., Mateescu, R.: An overview of CADP 2001. European Assoc. for Software Science and Technology (EASST) Newsletter **4** (2002) 13–24
37. ERTMS: Project Website (Jan 20 2006) `http://ertms.uic.asso.fr/etcs.html`.
38. ARP4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. Aerospace Recommended Practice, Society of Automotive Engineers, Detroit, USA (1996)
39. AVACS::S3: Project website (Jan 27 2006) `http://www.avacs.org/s3`.

A Kanban

For generating symbolic LTS representations, we have applied the stochastic process algebra tool CASPA (see [20]) from which we extracted symbolic, i.e. BDD, representations for a Kanban system [35]. The process algebraic description of the Kanban system consists of four parallel processes that are synchronized among each other. The system itself can be parameterized by a maximum number of *workpieces* that each process can handle.
We performed two series of experiments with two different configurations of the Kanban system. In the first configuration we have hidden all internal process actions, such that only the synchronization actions were visible. This kind of configuration could be of interest when inter-process communication only is analyzed. In the second configuration we have hidden all actions but the actions that are related to the first of the four processes. The motivation for this configuration is that one only wants to analyze the first process and likes to ignore the others.

```
int ini = 6;

kanban := (hide tredo1, tok1, tback1, tredo2, tok2, tback2,
                tok3, tback3, tredo3, tredo4, tok4, tback4 in
          (k1(ini,0,0,0) |[tsync1_23]|
          (k2(ini,0,0,0) |[tsync1_23,tsync23_4]|
           k3(ini,0,0,0))) |[tsync23_4]|
           k4(ini,0,0,0)
          )

k1(a [ini],x [ini],y [ini],z [ini]) :=  [a>0] -> (in1,1) ; k1(a-1,x+1,y,z)
                                        [x>0] -> (tredo1,1) ; k1(a,x-1,y+1,z)
                                        [x>0] -> (tok1,1) ; k1(a,x-1,y,z+1)
                                        [y>0] -> (tback1,1) ; k1(a,x+1,y-1,z)
                                        [z>0] -> (tsync1_23,1) ; k1(a+1,x,y,z-1)

k2(w [ini],x [ini],y [ini],z [ini]) :=  [w>0] -> (tsync1_23,1) ; k2(w-1,x+1,y,z)
                                        [x>0] -> (tredo2,1) ; k2(w,x-1,y+1,z)
                                        [x>0] -> (tok2,1) ; k2(w,x-1,y,z+1)
                                        [y>0] -> (tback2,1) ; k2(w,x+1,y-1,z)
                                        [z>0] -> (tsync23_4,1) ; k2(w+1,x,y,z-1)

k3(w [ini],x [ini],y [ini],z [ini]) :=  [w>0] -> (tsync1_23,1) ; k3(w-1,x+1,y,z)
                                        [x>0] -> (tredo3,1) ; k3(w,x-1,y+1,z)
                                        [x>0] -> (tok3,1) ; k3(w,x-1,y,z+1)
                                        [y>0] -> (tback3,1) ; k3(w,x+1,y-1,z)
                                        [z>0] -> (tsync23_4,1) ; k3(w+1,x,y,z-1)

k4(w [ini],x [ini],y [ini],z [ini]) :=  [w>0] -> (tsync23_4,1) ; k4(w-1,x+1,y,z)
                                        [x>0] -> (tredo4,1) ; k4(w,x-1,y+1,z)
                                        [x>0] -> (tok4,1) ; k4(w,x-1,y,z+1)
                                        [y>0] -> (tback4,1) ; k4(w,x+1,y-1,z)
                                        [z>0] -> (tout4,1) ; k4(w+1,x,y,z-1)
```

Fig. 2. CASPA input for the first configuration of the Kanban system.

The input files for both configurations are given in Figures 2 and 3. Please note that the variable *ini* denotes the number of workpieces along which the

```
int ini = 6;

kanban := (hide tredo2, tok2, tback2, tok3, tback3, tredo3,
                tredo4, tok4, tback4, tsync23_4 in
          (k1(ini,0,0,0) |[tsync1_23]|
          (k2(ini,0,0,0) |[tsync1_23,tsync23_4]|
           k3(ini,0,0,0))) |[tsync23_4]|
           k4(ini,0,0,0)
          )

k1(a [ini],x [ini],y [ini],z [ini]) :=  [a>0] -> (in1,1) ; k1(a-1,x+1,y,z)
                                        [x>0] -> (tredo1,1) ; k1(a,x-1,y+1,z)
                                        [x>0] -> (tok1,1) ; k1(a,x-1,y,z+1)
                                        [y>0] -> (tback1,1) ; k1(a,x+1,y-1,z)
                                        [z>0] -> (tsync1_23,1) ; k1(a+1,x,y,z-1)

k2(w [ini],x [ini],y [ini],z [ini]) :=  [w>0] -> (tsync1_23,1) ; k2(w-1,x+1,y,z)
                                        [x>0] -> (tredo2,1) ; k2(w,x-1,y+1,z)
                                        [x>0] -> (tok2,1) ; k2(w,x-1,y,z+1)
                                        [y>0] -> (tback2,1) ; k2(w,x+1,y-1,z)
                                        [z>0] -> (tsync23_4,1) ; k2(w+1,x,y,z-1)

k3(w [ini],x [ini],y [ini],z [ini]) :=  [w>0] -> (tsync1_23,1) ; k3(w-1,x+1,y,z)
                                        [x>0] -> (tredo3,1) ; k3(w,x-1,y+1,z)
                                        [x>0] -> (tok3,1) ; k3(w,x-1,y,z+1)
                                        [y>0] -> (tback3,1) ; k3(w,x+1,y-1,z)
                                        [z>0] -> (tsync23_4,1) ; k3(w+1,x,y,z-1)

k4(w [ini],x [ini],y [ini],z [ini]) :=  [w>0] -> (tsync23_4,1) ; k4(w-1,x+1,y,z)
                                        [x>0] -> (tredo4,1) ; k4(w,x-1,y+1,z)
                                        [x>0] -> (tok4,1) ; k4(w,x-1,y,z+1)
                                        [y>0] -> (tback4,1) ; k4(w,x+1,y-1,z)
                                        [z>0] -> (tout4,1) ; k4(w+1,x,y,z-1)
```

Fig. 3. CASPA input for the second configuration of the Kanban system.

descriptions are parameterized. The main difference between both input files is the actions that are hidden (using the `hide`-operator).

B ETCS

B.1 Overview

This case study is based on the European Train Control System (ETCS) (see [37]). The model especially focuses on the issues arising from failures in the communication between trains and the RBC.
One of the main concerns during the construction of this model was scalability. The current implementation can handle arbitrary many trains. The instances of the model that have been analyzed within this work use one, two and three trains.

Architecture. Some simplification were necessary in order to arrive at a model that is suitable for our current tool chain. We only consider one track controlled by one Radio Block Center (RBC) and do not model the change of tracks of the trains as this would imply changes in the architecture.

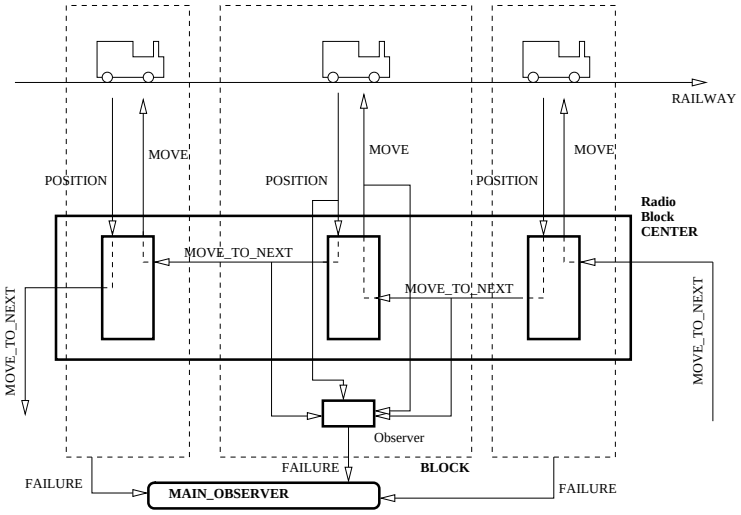

Fig. 4. Architecture.

An overview of the system architecture is given in Figure 4 where n trains are moving on the track. Each train communicates with the RBC: the RBC is divided into n local block, each block is responsible for the communication with one dedicated train.

The RBC operates as follows, it receives the current position of each moving train. To authorize a train to move on, it sends an authorization message. The idea is that the RBC only sends a moving authorization after it got the position from the previous train. Since a train is only allowed to send its new position if it is moving, each train can only move if the previous trains did already move before.

With the assumptions that a train can either move with full speed or doesn't move at all and that the track the train is moving on is constant, the safety requirement becomes much simpler and no calculation of the exact train position is necessary. The train is safe as long as it only moves after receiving the authorization (as a MOVE-message) from the RBC and the RBC only sends a new authorization after it got a POSITION-message from the previous train.

This requirement is monitored by an observer that checks the right sequence of control-messages. Each RBC-block has its own observer that reports a failure to a main observer. These observers also control that the trains don't stop if the track is free.

Several failure have been taken into account that can lead to faulty and unsafe behavior. For example the communication between the RBC and the trains can be lost.

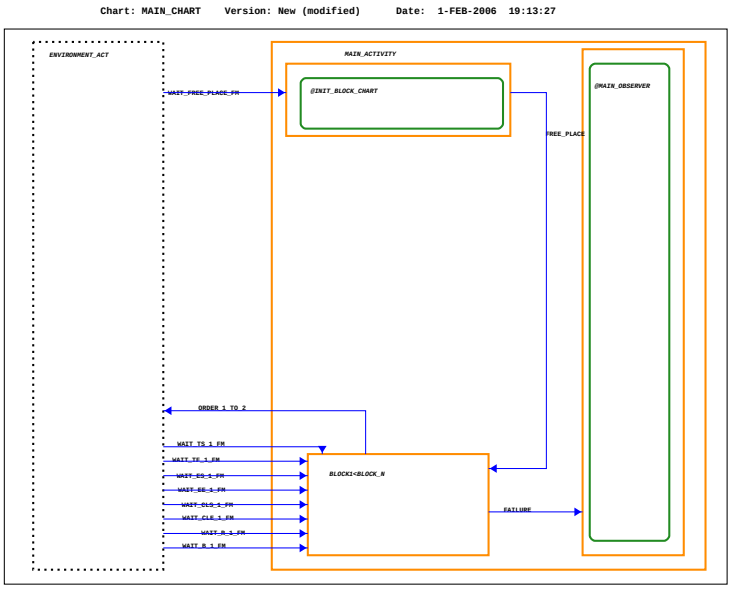

Fig. 5. Main chart.

B.2 Model Description.

Main Chart. The `MAIN_CHART` comprises the trains, a `MAIN_OBSERVER`, and an initializer, responsible for giving the first train on the track the authority to move on (see Figure 5). The initializer simply fires the `FREE_PLACE` event, if the condition `WAIT_FREE_PLACE_FM` from the environment is set. The `MAIN_OBSERVER` waits for the `FAILURE` event from a train. The only thing it has to do is to catch all failures from the trains and switch to its `FAILURE` state.

Structure of Trains. There is one activity chart for each train that is split into three parallel statecharts (see Figure 6). The first represents the RBC-communication for this train, the second the train itself, and the third the observer. Each activity chart gets an input `MOVE_FROM_PRED`[11] from the activity chart of the train in front of it, and sends an output `MOVE_TO_NEXT` to the activity chart belonging to the train behind it. Furthermore every activity charts gets eight different conditions as input, representing the environmental behavior, which can not directly be influenced by the controller.

[11] On the highest level these events are represented in variables named ORDER_n_TO_m, with $n, m \in \{1, 2, 3\}$ belonging to the number of the trains.

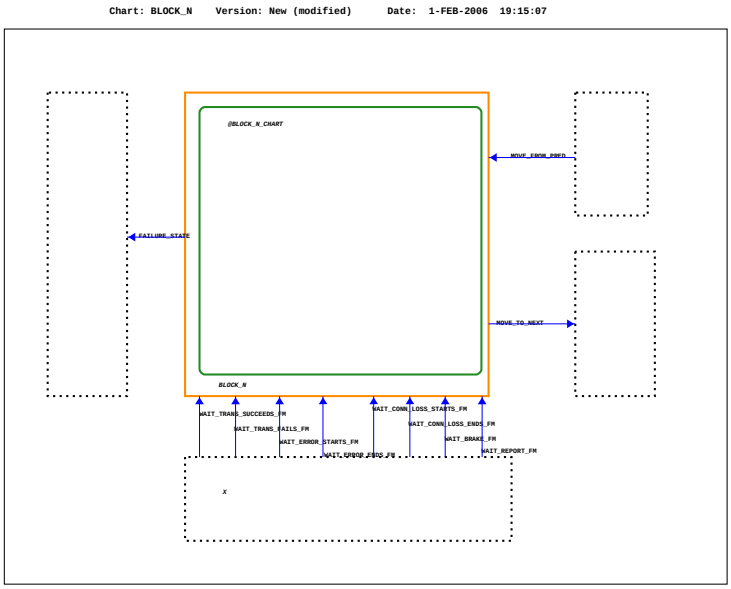

Fig. 6. Interface of a train.

FREE_PLACE This condition should be set, if there is a free place at the start of the track section.

TRANS_SUCCEEDS Will be set, if the transmission from the train to the RBC is successful.

TRANS_FAILS Will be set, if the transmission from the train to the RBC is not successful.

ERROR_STARTS Will be set, if an internal error inside the RBC occurs. The normal way to handle such an error is, to restart the RBC.

ERROR_ENDS Will be set, if an internal error inside the RBC is repaired (for example, if the RBC was restarted).

CONN_LOSS_STARTS Will be set, if the connection between the RBC and the train is lost.

CONN_LOSS_ENDS Will be set, if the connection between the RBC and the train is restored. It is assumed, that the RBC is in its idle state, when the connection is reestablished.

BRAKE Will be set, if the train did not receive a new moving authorization, for some time.

REPORT Will be set, when all necessary information for a new position report to the RBC is collected.

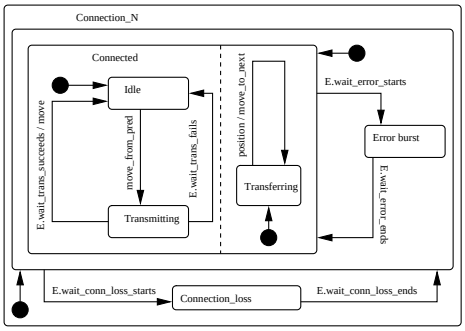

Fig. 7. Connection Train–RBC.

RBC Communication. Most of the environmental behavior influences the communication behavior between the train and the RBC (see Figure 7).

Normally the RBC part, responsible for the train, is idle (state `IDLE`). If it receives a position information from the train in front (event `MOVE_FROM_PRED`), it tries to transmit a moving authorization. Depending to the environmental behavior, this either fails, or succeeds (conditions `TRANS_FAILS` or `TRANS_SUCCEEDS`). The moving authorization will be submitted as an event (`MOVE`) to the parallel state which represents the train. If a train sends its position report to the RBC, an event (`MOVE_TO_NEXT`) is send to the next.

There are two types of errors, that can cause the communication to fail. If the condition `ERROR_STARTS` is set, no communication between the train and the RBC is possible, due to an internal error of the RBC. The same applies when the condition `CONN_LOSS_STARTS` is set, indicating a connection loss. The loss of connection has a higher priority, because it does not matter, whether the RBC works correctly or not, if there is no connection. To return into normal (connected) behavior, the conditions `ERROR_ENDS` or `CONN_LOSS_ENDS` have to be set.

Train. The train is divided in two parallel parts (see Figure 8) The first controls the moving of the train. After getting a `MOVE` event from the RBC communication part, the train is in the `MOVING` state until the `BRAKE` condition is set. After that the train waits in the `BRAKING` state until a new moving authorization is given. The second part controls the position reports. If the parallel chart is in state `MOVING`, a new position is reported (via the `POSITION` event). After that, the train has to wait in the state `REPORT_SENT` for a new `REPORT` event, which indicates, that all necessary information for a new report are collected. Then it changes to the `REPORT_READY` state, from which it can send a new position report (provided that it is in the `MOVING` state).

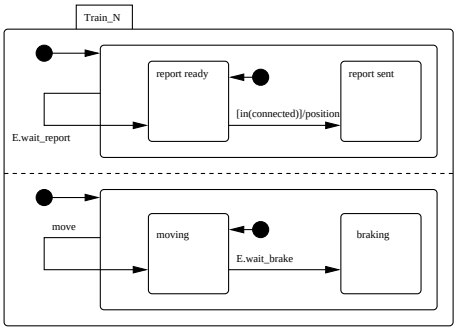

Fig. 8. Train.

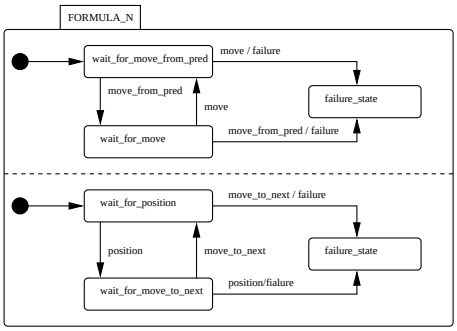

Fig. 9. Observer.

Observer. The observer has two parallel states, one observing the correct sending of a moving authorization, and the other observing the correct sending of a position report (see Figure 9).
The moving authorization is observed by making sure, that the `MOVE_FROM_PRED` event is followed by the `MOVE` event. This is done by two exclusive states (`WAIT_FROM_PRED` and `WAIT_FOR_MOVE`). If one of the events is set without the other the chart switches to a failures state, and fires a `FAILURE` event to the `MAIN OBSERVER`. The position report is observed in a similar way. Here the events `POSITION` and `MOVE_TO_NEXT` have to occur alternating. If not the chart switches in a failure state again, and the `FAILURE` event is fired to the `MAIN OBSERVER`.

Scalability. Scalability in this case means chaining blocks in STATEMATE. Whenever a new block is added to the main activity chart, one just has to draw the 4 information flows and enter the corresponding variables on every information flow (three events and 1 "*Channel_form_environment*"). As the main observer is not generic, the variable from one block to this observer must be called `FAILURE`.

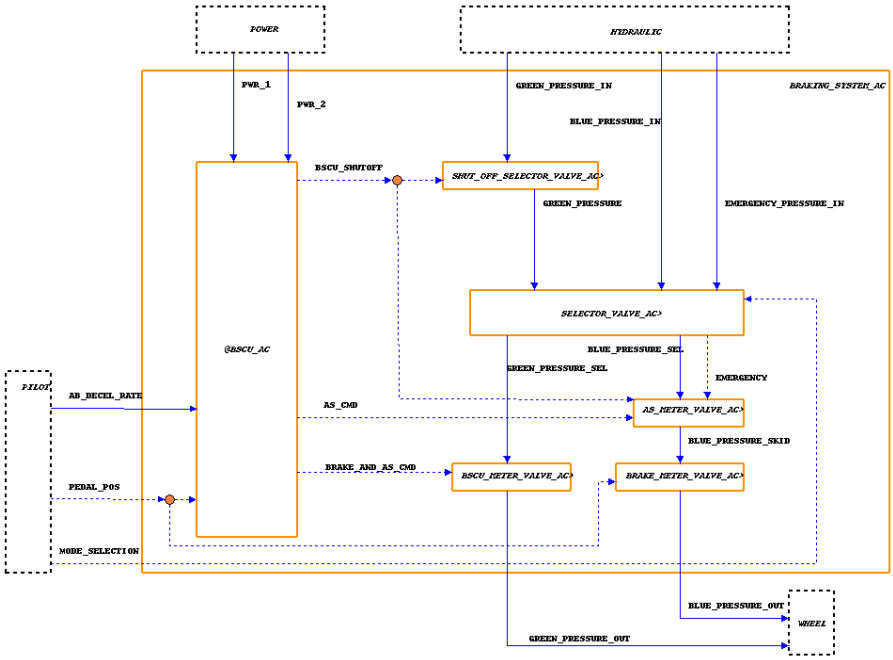

Fig. 10. The main activity chart of the braking system.

C Braking System

C.1 Model Description

The braking system is taken from the case study used in the ARP 4761 [38]. The job of the system is to provide hydraulic pressure to the wheels of the aircraft, whenever the pilot pushes the braking pedals. The auto-braking mode, where the pilot specifies a deceleration rate and the braking system computes the necessary braking commands itself, is not considered.

The system consists of three redundant hydraulic pressure lines, various valves and a computer called BSCU (*Braking System Command Unit*), which computes braking and anti-skid commands based on the pilot input. The structure of the system is shown in Figure 10. The inputs to the system are supplied by three external activities. `POWER` supplies electrical power needed by the BSCU via two redundant lines `PWR_1` and `PWR_2`. Hydraulic pressure (`HYDRAULIC`) is supplied from three independent lines, the green or normal line (`GREEN_PRESSURE_IN`), the blue or alternate line (`BLUE_PRESSURE_IN`) and the emergency line (`EMERGENCY_PRESSURE_IN`). Note that this is a small difference to the original system given in the ARP, as the emergency pressure is supplied from inside the system there. The remaining inputs are given by the `PILOT`. These are the braking command specified via the position of the braking pedals (`PEDAL_POS`),

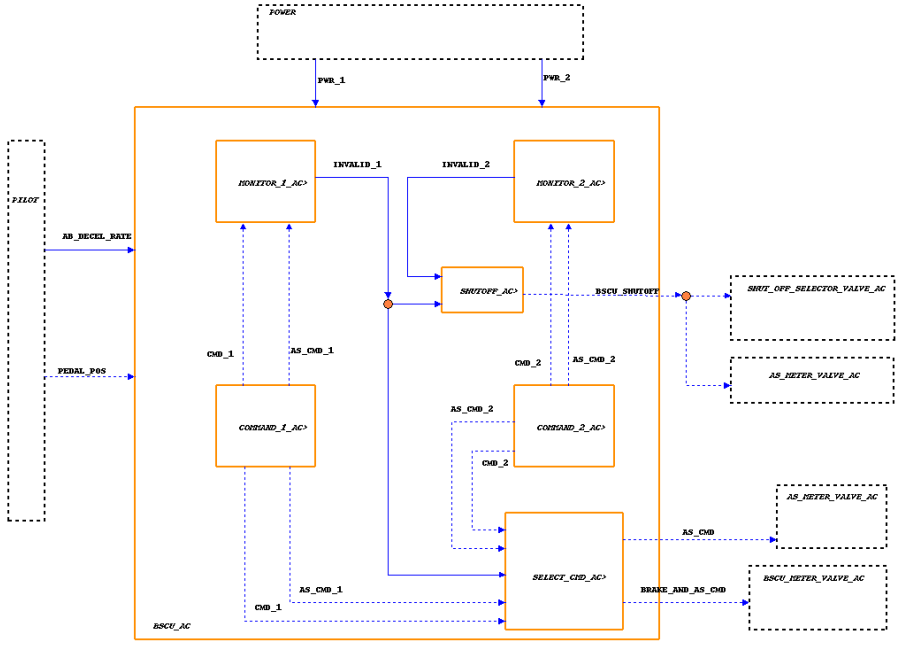

Fig. 11. The implementation of the braking system control unit.

the manual mode selection (`MODE_SELECTION`, see below) and the deceleration rate for the auto-braking (`AB_DECEL_RATE`, currently unused). The output of the system is the hydraulic pressure applied to the wheels. Pressure can be supplied via two redundant lines, `GREEN_PRESSURE_OUT` and `BLUE_PRESSURE_OUT`. These lines are driven by the three different operational modes, corresponding to the three input pressure lines.

The *normal* mode is driven by the green pressure line and is completely controlled by the BSCU. When the BSCU or the green pressure line fails, the *alternate* mode engages. Pressure is supplied via the blue hydraulic line, basic braking commands are now directly controlled by the pilot. The BSCU will add anti skid commands if it is working. When the blue pressure fails, the emergency pressure kicks in which also drives the blue output pressure. In emergency mode, anti skid is disabled, regardless of the BSCU state. Mode switches are controlled by the `SELECTOR_VALVE_AC`. It switches to the next available pressure line, when the currently active on fails. Switching is monotonic in this version of the model – the system starts in normal mode and can only switch "down" to the next available mode, not back. When the BSCU stops operating, it sends a shutoff signal to the `SHUT_OFF_SELECTOR_VALVE_AC` valve, which inhibits the green pressure to reach the selector valve, thus deactivating the normal mode.

The implementation of the BSCU is shown in Figure 11. It consists of two redundant computation units (`COMMAND_1_AC` and `COMMAND_2_AC`), two monitoring

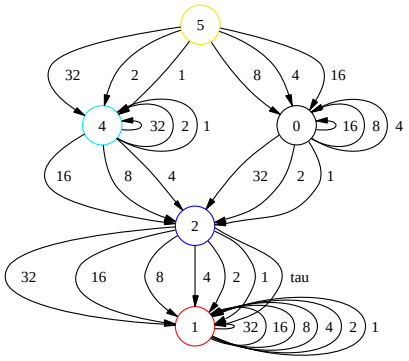

Fig. 12. The quotient for the BSCU shutoff.

units (`MONITOR_1_AC` and `MONITOR_2_AC`), the logic to chose between these two lines (`SELECT_CMD_AC`) and the shutoff monitor (`SHUTOFF_AC`). In the current implementation, each command unit simply copies the braking command given by the pedal position. The monitoring units compute a reference command in the same way and compare it to the output of the command units. As soon as those commands differ, the corresponding invalid signal is set (`INVALID_1` or `INVALID_2`). The switching unit selects the command computed by the first command unit, unless the `INVALID_1` signal is set. As soon as both invalid signals are set, the shutoff unit sets the shutoff signal.

C.2 Safety Analysis

There are two safety requirements which are considered in the analysis of the braking system. The first requirement under consideration is the shutoff of the BSCU. The following failure modes are considered for this analysis:

- failure of the command computation (`CMD_{1,2}`)
- failure of the anti skid command computation (`AS_CMD_{1,2}`)
- failure of the monitor command computation (`MON_{1,2}_CMD`)

The analysis generated the quotient shown in Figure 12.
The second safety requirement under consideration is the complete loss of braking commands. The failure modes under consideration are the same as above and additionally:

- failure of the invalid signals (`INVALID_{1,2}`),
- failure of the command selection switch (`SELECT_CMD_2`)
- failure of electrical power (`PWR_{1,2}`)
- failure of hydraulic pressure (`{GREEN,BLUE,EMERGENCY}_PRESSURE_IN`)
- manual mode selection (`MODE_SELECTION`)