On the formal verification of systems of synchronous software components (extended version)*

Henning Günther[1], Stefan Milius[1], and Oliver Möller[2]

[1] Institut für Theoretische Informatik,
Technische Universität Braunschweig, Germany
[2] Verified Systems International GmbH
Bremen, Germany

Abstract. Large asynchronous systems composed from synchronous components (so called GALS—globally asynchronous, locally synchronous—systems) pose a challenge to formal verification. We present an approach which abstracts components with contracts capturing the behavior by a mixture of temporal logic formulas and non-deterministic state machines. Formal verification of global system properties is then done transforming a network of contracts to model checking tools such as PROMELA/SPIN or UPPAAL. Synchronous components are implemented in SCADE, and contract validation is done using the SCADE Design Verifier for formal verification. We also discuss first experiences from an ongoing industrial case study applying our approach.

Keywords: formal verification, GALS systems, rely-guarantee, SCADE, SPIN, UPPAAL

1 Introduction

State-of-the-art safety critical systems are often composed of other distributed (component) systems (**system of systems (SoS)**). While industrial standards for the development of safety-critical software systems highly recommend formal model-based methods, application of those methods to SoS still remains a challenge when scalability to real industrial applications is concerned.

In this paper we report on work in progress concerning the development of an approach to modeling and verification of SoS that is innovative for the industrial practice and addresses the scalability problem. In our approach the nodes of a distributed system consist of controllers performing specialized tasks in hard real time by operating cyclically and in a synchronous way. For such a controller the model-based approach of SCADE[3] is an attractive solution providing code generation and good support for model simulation and (formal) verification. But for a distributed system, a synchronous implementation is neither realistic nor desirable. Hence, we focus on the model-based

* This work was developed during the course of the project "Verifikation von Systemen synchroner Softwarekomponenten" (VerSyKo) funded by the German ministry for education and research (BMBF).

[3] SCADE is developed and distributed by Esterel Technologies:
www.esterel-technologies.com

development and analysis of asynchronously communicating embedded control systems that are composed from components that operate synchronously; this is known as a GALS (**globally asynchronous – locally synchronous**) architecture; this goes back to Chapiro [8], and it is the preferred solution for complex safety relevant control tasks.

The main idea to address the complexity issues of GALS systems is to provide for each synchronous component an abstract model in the form of a **contract** that can be locally verified for the component (e. g. by SCADE Design Verifier, the formal verification engine of SCADE). The network of component contracts then forms an **abstract GALS model** against which a system requirement (called **system-level verification goal**) can be formally verified. This is done by a model transformation of the abstract GALS model into an appropriate formalism/tool for model checking—we use PROMELA/SPIN and UPPAAL, respectively, for model checking system-level verification goals.

Integrating synchronous components within an asynchronous environment using the GALS approach and using abstraction to handle system's complexity are not new ideas. What *is* new in our work is the combination of GALS verification with the idea of abstraction by contracts and its application to networks of synchronous SCADE models. Hence, since SCADE is an industrial strength tool for synchronous controller software, our framework contributes towards closing the methodological gap between the applicability of formal verification for single controllers and asynchronously composed systems of such controllers in an industrial context.

In addition, the previous work on GALS systems pertains to systems whose components were designed to interact synchronously but are later integrated asynchronously. In our work we assume that GALS systems are designed to consist of synchronous components that are intended to be composed asynchronously (GALS systems by design). We introduce a new specification language for GALS systems, and we design and implement model transformations between our language and appropriate model checking tools (PROMELA/SPIN, SCADE Design Verifier and UPPAAL). These are parts of a larger framework for (formal) verification of GALS systems that also contains higher level, user-friendly and domain specific (graphical) languages as well as methods and tools for test automation and analysis. These other aspects of the framework cannot be presented within the page constraints of this paper. More details on them can be found in the technical report [33] and in [17]. We also do not discuss a systematic way to derive suitable contracts for given components—this can be a challenging task in practise, but we leave the solution of this problem for future work.

We begin in Sec. 2 with a discussion of our system level verification approach. In Sec. 3 we introduce our modeling language for GALS systems—the GALS translation language (GTL). Next we briefly describe the transformation algorithms used for local and global verification of GALS systems (Sec. 4) and we provide a benchmark for the verification back-ends using a simple example of a GALS system (Sec. 5). Finally, we report about first experiences of our tools on an industrial case study in Sec. 6.

1.1 Related Work. Numerous publications are devoted to combining synchrony with asynchrony and the verification of GALS systems. For example, Doucet et al. [11] describe how C-Code generated from synchronous components in SIGNAL is integrated in PROMELA abstracting the communication framework by FIFO channels. Thivolle

and Garavel [12] explain the basic idea of combining a synchronous language and an asynchronous formalism, and they also show how synchronous components can be integrated into an asynchronous verification tool and demonstrate this with one simple example. In these works components are not abstracted by contracts as in our approach but synchronous models are directly integrated in an asynchronous formalism. However, the results from our case study clearly indicate that component abstraction is necessary.

A different approach follows Milner's result [26] that asynchrony can be encoded in a synchronous process calculus, see e. g. [20, 22, 28], and the tool Model build [5, 6] as well as the Polychrony workbench [25]. A disadvantage of these approaches is that asynchrony and non-determinism are not built-in concepts in the underlying formalisms and so verification tools may not be optimized for asynchronous verification.

Other approaches extend synchronous formalisms in order to deal with some degree of asynchrony, e. g., Multiclock Esterel [29] or Communicating Reactive State Machines [30, 31]. Again, components are not abstracted in these approaches, and according to [12]: "such extensions are not (yet) used in industry".

Using contracts as specifications for parts of a program or system is also not a new idea; see for example work on rely/guarantee logic [23]. Abstracting system components by contracts appears recently, for example, in [15, 14] and in [7]. The former work uses component contracts in the form of time-annotated UML statecharts. So this approach does not deal directly with synchronous components or GALS systems. In addition, component contracts cannot be specified by LTL formulas as in our framework. The latter work [7] describes a way to use contracts to specify the behaviour and interactions of hardware components. The focus is on the verification of contracts while our work also considers formal verification of system-level verification goals of composed contract-systems.

Alur and Henzinger [3] treat the semantics of asynchronous component systems, their *reactive modules* can be used to give a semantics to our GALS system specifications (see Sec. 3.2 below). Reactive modules are also the basis for the tool Mocha [1] which uses Alternating Temporal Logic (ATL) as a specification language for system requirements. In our approach the specification language for contracts and global verification goals is separate from the synchronous language in which components are implemented. So our framework is more flexible—it allows to easily exchange the synchronous language for components and it also allows to change the analysis tools used for formal verification.

Clarke et al. describe an automatic approach to generating contract abstractions [9]. We did not apply this technique in our framework (yet) because we believe there are several difficulties with this approach: it can only generate abstractions with the same expressive power as regular languages, while our approach can also handle LTL abstractions. Also, the number of iterations needed for finding the abstraction might outweigh the performance gains of the abstraction itself. But this still needs to be investigated systematically in the future.

To sum up, the various ingredients (contracts for abstraction, synchronous verification, GALS systems) of our work are well-established in the literature. However, to the best of our knowledge these ingredients have not been brought together in this form for

the verification of GALS systems of synchronous SCADE models, and it is this gap we intend to fill with our work.

2 Verification of GALS systems

In this section we explain our general approach to system level verification of GALS systems. Within our framework system verification proceeds along the following lines:

(1) System-level verification goals Φ are specified as (timed) LTL formulas expressing the desired behavior of the complete GALS system.

(2) The behavior of each synchronous component M is abstracted by its contract C. The contract C contains an interface description of M, and to specify component behavior we use a mixture of LTL formulas ("implicit modeling style") and non-deterministic state machines ("explicit modeling style"). Local verification then ensures that each concrete component *implements its contract* which means that the traces matching the concrete component are a subset of the traces matched by the contract, written: $M \preceq C$.[4]Optionally, one may specify *guaranteed behavior* represented by additional LTL formulas. Guaranteed behavior are assertions of specific behavioral situations ("use cases") which have already been exhaustively verified on component level. This redundant information can be used during (manual) system-level validation to uncover flaws in contract specifications or verification goals; for more details see [33].

(3) From the specification of all component contracts and their composition to a network of components we derive an *abstract* GALS model C_G; this model exhibits every possible behavior allowed by the contracts.

(4) The assertion *"system satisfies Φ"*, i.e. $M_G \models \Phi$ for the network M_G of concrete components is verified by property checking $C_G \models \Phi$ instead.

In this approach, the handling of verification failures (i. e., the formal verification of $C_G \models \Phi$ produces a failure trace π) deserves attention: Because the abstract network has (in general) more traces than the concrete one, it follows that the failure trace π of the abstract network is possibly not a trace of the concrete one. We call this a *false negative* and it can be uncovered by running a simulation of the concrete network, restricted to traces where the observable behavior matches π.[5] If this simulation is successful, we know that the failure trace π is indeed a witness to a failure in the concrete network M_G. Otherwise, we can conclude that at least one of the contracts is too weak and has to be strengthened to achieve successful verification.

Finally, notice that a *false positive*, i. e., the formal verification of $C_G \models \Phi$ succeeds while $M_G \not\models \Phi$ for the concrete network M_G of components, can happen because of an inconsistency of a contract C with its concrete model M. However, this cannot happen if local verification of $M \preceq C$ in item (2) above is successful. As usual, we assume that Φ correctly formalizes its corresponding informal requirements.

[4] Here we borrow notation from Alur's and Henzinger's reactive modules [3], cf. Sec. 3.2.

[5] This is possible because the abstracted GALS model C_G operates on the complete concrete interfaces specified for each component M, so that abstraction only introduces more general behavior, but not abstracted data.

3 GALS Translation Language (GTL)

In this section we present our specification language for GALS systems. We show how to specify for each component M its contract C, how components are instantiated and composed to a GALS system and how system-level verification goals are specified.

For illustration we use a simple mutual-exclusion specification in which three clients compete for a single resource (see Fig. 1). Each client is initially in its non-critical section (state `NC`), may then want to acquire the resource (state `ACQ`), will then enter its critical section (state `CS`) and must leave this section again after at most 5 synchronous cycles (state `REL`). A server component which communicates with each client has to ensure that only one client gets access to the resource at any time. We consider two classic verification goals: the first one is a safety condition stating that at no point in time more than one client is in the critical section, and the second one a liveness condition stating that no client stays forever in its critical section.

3.1 Syntax. Each synchronous component type is introduced with a "model"-declaration (lines 1 and 25 of Fig. 1). This declaration states the synchronous formalism in which the component is implemented (in this example as SCADE models), the unique name for reference in the GTL-file and a list of parameters which are needed to extract the implementation (e.g., the location of the file in which the model is implemented or its path in a library of components).

The interface of the component is declared by specifying input-, output- and local variables (see lines 2–3 and 26–27). While the input- and output-variables must be identical to the in- and outputs of the concrete (SCADE) component, the local variables may be different.[6] The GTL supports a wide range of types, including integers, booleans, enumerations, arrays and tuples.

Initial values for the interface variables may be specified (lines 5 and 29), and for each component type its cycle-time can be specified in (milli- or nano-)seconds.

Contracts for the model are also specified inside the model declaration. Each contract can be either an automaton or an LTL formula. Automaton-contracts are a list of states containing formulas which must hold for them and transitions into other states (lines 6–22); the LTL formula in line 23 specifies that the critical section is left within 5 cycles, and the server component is also specified by an LTL formula. One can form multiple instances of models (lines 38–41), and instances may add contracts to their component type. Guaranteed behavior can be specified by LTL formulas following the keyword `guaranteed`.

To enable communication between instances, a `connect` statement is used to link an output variable of one component to the input of another component of the same type (lines 42–44).

Finally, verification goals are specified as LTL formulas. These formulas can use all in- and output variables of any component in the system (lines 45–52). It is usually unclear which component of a composed GALS system makes a step in order for the system to reach its successor state, so verification goals can use the temporal connectives $\mathsf{next}[t]\,\phi$, $\mathsf{finally}[t]\,\phi$ and $\phi\ \mathsf{until}[t]\ \psi$, where t is a specified time.

[6] Note that it is not necessary to declare input and output variables if a SCADE component is used as the type information can be extracted from it.

```
model[scade] client("mutex.scade","Mutex::Client") {
  input bool proceed;
  output enum { NC, ACQ, CS, REL } st;
  cycle-time 5ms;
  init st 'NC;
  automaton {
    init state nc {
      st = 'NC or st = 'REL;
      transition acq;
      transition nc;
    }
    state acq {
      st = 'ACQ;
      transition[proceed] cs;
      transition[!proceed] acq;
    }
    state cs {
      st = 'CS;
      transition nc;
      transition cs;
    }
  };
  always (st = 'CS => (st = 'CS until[5cy] st = 'REL));
}
model[scade] server("mutex.scade","Mutex::Server") {
  input enum { NC, ACQ, CS, REL }^3 procstates;
  output bool^3 procouts;
  cycle-time 1ms;
  init procouts [false,false,false];
  always (procstates[0] = 'ACQ and procstates[1] != 'CS
          and procstates[2] != 'CS and procouts = [true,false,false])
      or (procstates[1] = 'ACQ and procstates[0] != 'CS
          and procstates[2] != 'CS and procouts = [false,true,false])
      or (procstates[2] = 'ACQ and procstates[0] != 'CS
          and procstates[1] != 'CS and procouts = [false,false,true])
      or (procouts = [false,false,false]);
}
instance client c0;
instance client c1;
instance client c2;
instance server s;
connect c0.st s.procstates[0];
...
connect s.procouts[2] c2.proceed;
verify {
  always (c0.st = 'CS => !(c1.st = 'CS or c2.st = 'CS));
  always (c1.st = 'CS => !(c0.st = 'CS or c2.st = 'CS));
  always (c2.st = 'CS => !(c0.st = 'CS or c1.st = 'CS));
  always (c0.st = 'CS => finally[30ms] c0.st = 'REL);
  always (c1.st = 'CS => finally[30ms] c1.st = 'REL);
  always (c2.st = 'CS => finally[30ms] c2.st = 'REL);
}
```

Fig. 1. GTL specification of the mutual exclusion example

3.2 Semantics of GTL specifications. Currently, the semantics of GTL specifications is purely transformational given by the model transformations of Sec. 4. We shall now sketch an idea how to define a formal semantics of GTL specifications using *reactive modules* of Alur and Henzinger [3], and we will use notions and notation from the theory of reactive modules to give an informal argument to substantiate our claim that our approach to formal verification of GALS systems is correct. We leave the task of working out the technical details of the proposed formal semantics for further work.

Our idea is to interpret every instance C of a component in a GTL specification as a reactive module with its input, output and local variables identical to the ones declared in the contract of C. The corresponding reactive module has exactly one update block which updates every output variable of the component. To every initialization declaration in C there is a corresponding initial action in the associated reactive module.

An automaton that is part of C's contract is translated by converting it from a mixed Moore/Mealy automaton to a pure Mealy-automaton and then introducing a fresh local variable which is used to store the current state of the automaton. For each transition from state s_1 to s_2 with the condition c, the following update action is added to the corresponding reactive module: $s = s_1 \wedge c \rightarrow s' = s_2$.

LTL contract formulas of C are translated by converting them into Büchi automata using the algorithm of Gastin and Oddoux [13]. To represent the resulting Büchi automaton within a reactive module one needs to add weak-fairness constraints to all transitions leading into a final state to ensure the preservation of the correct LTL-semantics.

The `connect` statements of the given GTL specifications are handled by using the same name for the connected in- and outputs in the associated reactive modules.

In this way an abstract network of the contracts $C_1, \ldots, C_n$ defined with a GTL specification corresponds to the composition

$$C_G = S \| C_1 \| \cdots \| C_n \tag{3.1}$$

on the level of reactive modules, where S is an (automatically generated) *scheduler*, i. e. a reactive module that ensures the fair synchronous execution of the C_i: all component start at the same time and then proceed according to their cycle times. Notice here that we abuse notation and write C_i for the reactive modules corresponding to the abstract components.

Similarly, any concrete SCADE model M of a component can be interpreted as a reactive module, and the network of the concrete components $M_1, \ldots, M_n$ then corresponds to the composition

$$M_G = S \| M_1 \| \cdots \| M_n, \tag{3.2}$$

where S is the same scheduler as above (again, abusing notation).

Of course, one can easily interpret LTL formulas over reactive modules, and since the scheduler component S introduces a global timer variable one can also interpret timed LTL formulas as follows: for a path $\pi = s_0, s_1, s_2, \ldots$ of an (abstract or concrete) network of components N we have (a) $\pi \models \mathsf{next}[t]\, \phi$ if ϕ holds for every state s_i in π reachable from s within time t, (b) $\pi \models \mathsf{finally}[t]\, \phi$ if there exists a state s_i in π reachable within time t in which ϕ holds, and (c) $\pi \models \phi\ \mathsf{until}[t]\ \psi$ if $\pi \models \phi\ \mathsf{until}\ \psi$ and there exists a state s_i in π reached within time t in which ψ holds.

As explained in Sec. 2, to verify $M_G \models \Phi$ for a verification goal Φ one now proceeds as follows: (1) one verifies by local verification that each concrete model implements its contract; in the notation of the corresponding reactive models: $M_i \preceq C_i$ for all $i = 1, \ldots, n$; (2) one verifies that the composed abstract model satisfies the verification goal: $C_G \models \Phi$. To see that this implies the desired result one argues as follows: firstly, by the compositionality of $\preceq$ it follows that $M_G \preceq C_G$, and, secondly, since the latter essentially means that the traces of M_G are contained in the traces of C_G it clearly follows that any LTL formula Φ satisfied by C_G also holds for M_G.[7]

Finally, note that on the level of reactive modules a false negative is a trace π of C_G witnessing $C_G \not\models \Phi$, whereas π is not a trace of M_G.

4 Model transformations for verification of GALS systems

In this section we show how various model transformations implement local and global verification as outlined in the previous sections. We also explain how a method for detection of false negatives is implemented.

4.1 Local Verification. The purpose of local verification is to show that for each contract C and corresponding SCADE model M we have $M \preceq C$. This verification task is done by transforming the contract into *synchronous observer* nodes in SCADE (cf. [19, 21, 10]). Each LTL formula contained in the contract is first translated to a state machine using the translation algorithm described by Gastin and Oddoux [13]. Due to restrictions of the SCADE Design Verifier, it is only possible to use safety- or time-constrained liveness properties for this translation. As a result, for each abstract component C we obtain a set of automata. Each of those automata is transformed into a SCADE synchronous state machine (cf. [4]). This yields a set of observers that receive the inputs and the outputs of M and generate boolean flows whose conjunction signifies whether M implements its abstract component C. More precisely, the conjunction of the outputs of the observers is true in a cycle iff the outputs produced by M on the inputs in that cycle are contained in the possible outputs admitted by the contract C given the same inputs. Fig. 2 shows the synchronous observer generated from the contract of the client component in our mutex example.

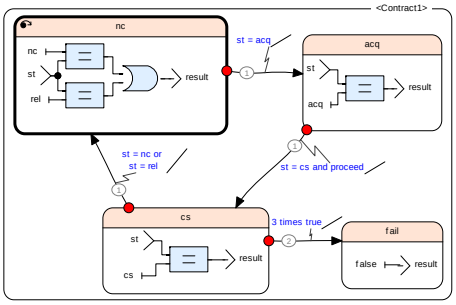

Fig. 2. Synchronous observer for client contract

4.2 Global Verification. In order to verify whether an abstract GALS model specified in GTL satisfies a verification goal Φ one generates from the GTL specification a PROMELA model C_G that behaves like the composite model from (3.1). To this end each contract is transformed into a set of automata as described above, and a PROMELA

[7] Note that this works because LTL is defined on paths. For CTL, which works on trees, this argument does not work.

process for the contract is created by forming the product automaton. The different processes asynchronously communicate via shared variables that correspond to the connections of inputs and outputs of the synchronous component; for each `connect` statement in the GTL model one shared variable is generated in the PROMELA model. There is no buffering; component outputs of previous cycles are simply overwritten. As previously explained, our model transformation also creates a scheduler for the fair synchronous execution of the components. If the verification goal Φ is an ordinary LTL formula, it can simply be verified whether $C_G \models \Phi$ by using SPIN. If Φ contains temporal connectives, timers are introduced in the PROMELA model. For example, the formula $\phi \,\mathsf{until}[t]\, \psi$ is translated to

$$(\mathsf{c} := t) \wedge ((\phi \wedge \mathsf{c} \geq 0) \,\mathsf{until}\, (\psi \wedge \mathsf{c} \geq 0));$$

and a timer variable c is created which is initialized with time t. Each time a synchronous component (or rather its PROMELA process) makes a step, the timer c is decremented by the amount of time that has passed since the last step was performed by a (possibly different) synchronous component. For this translation to be sound, $\mathsf{until}[t]$-formulas must not be nested on the right-hand side.

The translation of the network of contracts in a GTL specification to UPPAAL works similarly. However, it is not possible in general to translate all verification goals since the supported logical language of UPPAAL is based on (timed) CTL [2]. However, if we restrict ourselves to so-called safety properties, translation to CTL is both sound and simple. While clearly inferior in the expressive power, this logical class of formulas is sufficient for many practical purposes. Currently, translation of safety properties has to be done manually.

4.3 Detection of False Negatives. We implemented a third transformation from GTL that can be used to validate verification results for an abstract GALS model C_G. Suppose we have $C_G \not\models \Phi$ for a verification goal Φ and the formal verification produces the failure trace π. If each component comes with a SCADE implementation, we can check whether this is a real failure trace or a false negative as follows: using the GTL specification one generates the concrete GALS model M_G in PROMELA behaving like the composite model in (3.2). This is done by composing the SCADE models of the components (together with the scheduler) by integrating the C-code generated from them. By using SPIN to simulate M_G on the inputs from π we can verify whether π is a trace of M_G.[8]

If so, we have found a real error, and one or several component implementations need to be corrected. To support this process one can project the global failure trace π on a local trace π_M for each component M, which can be used in the ensuing analysis: from each π_M one can generate a SCADE simulator script which can be used to correct the SCADE models of the components.

If the simulation finds that π is not a legal trace of the concrete GALS model M_G, then our verification result is a false negative, and one needs to analyze the contracts for weaknesses or inconsistencies.

[8] Again, this simulation is possible since C_G operates on the complete concrete interfaces specified by each component M.

For a concrete example of a false negative let us consider our mutex example. If we omit line 23 of the client contract, which prevents clients from staying forever in their critical section, the global verification of the second verification goal in Fig. 1 will yield a failure trace: a client remains forever in the critical section. However, this does not happen in the concrete model; the SCADE model of the client (not shown here) will leave its critical section after at most 5 cycles.

5 Benchmark: The Mutex Example

To evaluate the GTL transformation to back-end formalisms, we use the mutex example from Sec. 3 as a benchmark.

To this end a sequence of GTL files is generated by increasing the number N of client processes that compete for the critical section. The server will (only) grant access to the critical section to one requesting client (chosen non-deterministically), if no client is currently in the critical section.

The state space of the system grows exponentially with increasing N. The property that is model-checked for all instances is the previously mentioned classic safety condition: at no point in time more than one client is in the critical section. Since this is true, the complete state space has to be analyzed.

The GTL representation is transformed to a model representation for SPIN and UPPAAL, respectively. For SPIN, a verifier is (gcc-) compiled and executed with runtime options `-a` and `-m9999999k`. This guarantees exhaustive and maximally deep search. Other than that, none of the numerous optimization options of the tools are activated. We use the newest available (64bit-)releases of the tools.

Fig. 3 displays the time and memory consumption with increasing number N of clients. Unmapped N correspond to out-of-memory situations. After an initial offset, the resource usage shows a steady slope on the logarithmic scale, which corresponds to the exponential growth of the state space. Both SPIN and UPPAAL follow mainly the same slope, but maintain roughly constant distance, which corresponds to a constant *factor*. The time plot shows this better than the memory plot, since the latter operates with a basic offset of allocated main memory (up to $N = 5$, due to option `-m`).

Surprisingly, this factor is rather large: ≈ 53 for time without compiler optimizations (≈ 23 with full optimization) and ≈ 87 for memory usage. Possibly, UPPAAL profits substantially from the fact that only the reachable states have to be allocated at all, while SPIN does provide (hash-compressed) memory for the full state space. More details can be found in [27].

6 Case study

In the previous section we showed that our approach works on small academic examples. To see whether our method scales up to realistic systems we are currently working on an industrial case study—a level crossing system from the railway domain.

The level crossing consists of several components (traffic lights, supervision signals, barriers etc.). An overview of the architecture is given in Fig. 4. The components have been implemented as synchronous SCADE models, and are of medium complexity: Failures, recovery and supervision aspects are implemented in each component. A detailed informal description of the requirements of the level crossing system and its

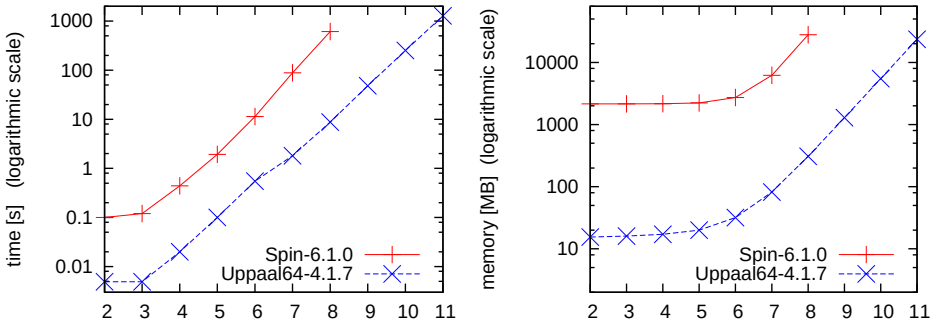

Fig. 3. Time- and memory consumption for exhaustive search for N clients; measured on a 2.80GHz Intel® Xeon® CPU with 24GB of main memory

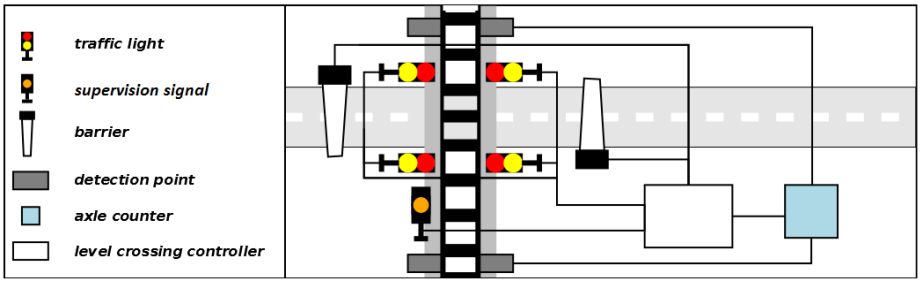

Fig. 4. Case study level crossing - system architecture

overall system architecture can be found in [32]. The implementation can be found at the VerSyKo project web page.[9] A main global requirement of the level crossing system is to protect the road traffic from the train traffic and vice versa. Without abstraction, the state space of the system is too large to be handled by model checkers like SPIN: an experiment to integrate the C-code generated from the SCADE models and using the model checker SPIN yields a too large state space. This outcome validates our expectation that it is necessary to reduce state space by providing abstractions of the local synchronous components using contracts.

As a next step, we have formulated contracts for each of the components of the level crossing system and used SCADE Design Verifier to prove the contracts correct. Unfortunately, for the level crossing controller, SCADE Design Verifier did not succeed in verifying our contract. The reason for this is yet unclear, but omitting one of the three automata from the contract yielded a verifiable contract. We suspect that the third automaton encodes a property that cannot be handled by the induction heuristics implemented in SCADE Design Verifier. However, the Debug Strategy of SCADE Design Verifier yielded no counterexamples unrolling the model up to depth 80. The results of the contract verification can be seen in Table 1, which also shows the complexity of both the SCADE model (estimated from the C-code generated from it) and the associated contract. The detection points in Fig. 4 do not appear because they are mere sensors without controller software.

[9] See `http://www.versyko.de`.

Component	Model complexity (no. of states)	Contract complexity (no. of states)	Verification time (s)
traffic light	$5.92 \cdot 10^{10}$	6	3.292
supervision signal	$1.54 \cdot 10^{4}$	4	5.054
barrier	$2.46 \cdot 10^{5}$	5	3.385
axle counter	$2.88 \cdot 10^{3}$	3	4.103
level crossing controller	$2.36 \cdot 10^{138}$	32	543.599[1]

[1] Verified up to depth 80 using bounded model checking.

Table 1. Contract verification times

For a first experiment with global verification we have formulated the main requirement mentioned above as a verification goal in GTL. Since we have not completed an automatic translation of GTL verification goals into UPPAAL's query language, we did not experiment with global verification using UPPAAL yet. Using SPIN resulted, as expected from our benchmark in the previous section, in complexity problems.

7 Conclusions and Future Work

We presented a framework for the formal verification of GALS systems built from synchronous SCADE models. Contacts are used as abstractions of concrete synchonous components in order to handle system complexity. The goal is to obtain an approach that can handle the formal verification of such systems in an industrial context. We also presented first results from an ongoing industrial case study.

Let us summarize our findings sofar. First of all, our first experiments confirmed our expectation that abstraction of components is necessary to handle the formal verification of global verification goals.

Our experience formulating contracts for the industrial case study showed that it can be non-trivial to define a correct and adequate abstraction that is qualified for model checking, and leads to a diagnostically conclusive result. It may be necessary to investigate the implementation in more depth. In addition, contracts may need to be tailored towards formal verification of a particular verification goal.

The local formal verification of contracts can be performed for small and medium sized components using SCADE Design Verifier. But for bigger components one may not be able to sucessfully complete formal verification. In such cases it is difficult to analyze the reason for this as information on the details of the verification algorithm of SCADE Design Verifier is not freely available. However, using the Debug Strategy of SCADE Design Verifier one may still perform bounded model checking to uncover errors in contract specifications and this way one can build trust in the correctness of the contract. In addition, we saw that verifying contracts helps improving the components' quality. For example, for the traffic light controller the contract validation has revealed a subtle error in the implementation. For two states in the SCADE model the transition priorities were wrong—in a situation where the model must proceed to a failure state it will instead transition to a different state, this error has been corrected in the implementation.

Our benchmark using SPIN and UPPAAL for global formal verification indicates that those two analysis tools do not scale to real industrial applications and this is confirmed by our experiment with the industrial case study. Since it is necessary to gener-

ate a scheduler component to facilitate the synchronous execution (in both SPIN and UPPAAL) of the abstract GALS models, the timing abstraction provided by timed automata in UPPAAL does not reduce the state space enough in order for this verification method to scale up.

Ongoing Work. To address the above mentioned complexity problems, we are now investigating a different and new approach using bounded model checking and a model transformation for global verification from GTL to an SMT-solver. Our first experiments using this approach indeed look promising and allow to check the absence of counterexamples to our global verification goals up to a fixed number of steps performed by the abstract GALS model. Details on this new approach and a more extensive investigation of the case study will be reported subsequently. More details on the current verification results can be found in [16].

Directions for future work include: (a) exploring possible alternatives to SCADE Design Verifier for local verification—an approach using bounded model checking with an SMT-solver similar to the KIND [24] model checker for LUSTRE will be investigated; (b) further investigations using bounded model checking for global verification will be made on our case study, in particular, the formalization of other requirements as global verification goals and the formulation of appropriate contracts for them; (c) from the point of view of applicability of our approach a systematic methodology how to find suitable abstractions of components and to formulate good contracts is highly desirable. At the moment this is a creative process that needs expertise both with the system under investigation and with the formal verification methods used in our framework. Concerning this point, it should be investigated in how far tthe CEGAR approach of [9] is applicable for automatic derivation of contracts.

Acknowledgments. We are grateful to ICS AG for providing the industrial case study, and to Axel Zechner and Ramin Hedayati for fruitful discussions and their support to formulate the contracts.

References

1. de Alfaro, L., Alur, R., Grosu, R., Henzinger, T., Kang, M., Majumdar, R., Mang, F., Meyer-Kirsch, C., Wang, B.: Mocha: Exploiting modularity in model checking (2000), `http://www.cis.upenn.edu/~mocha`
2. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Information and Computation 104(1), 2–34 (1993)
3. Alur, R., Henzinger, T.: Reactive modules. FMSD 15, 7–48 (1999)
4. André, C.: Semantics of S.S.M (safe state machine). Tech. Rep. UMR 6070, I3S Laboratory, University of Nice-Sophia Antipolis (2003)
5. Baufreton, P.: SACRES: A step ahead in the development of critical avionics applications (abstract). In: Proc. of HSCC. LNCS, vol. 1569. Springer-Verlag, London, UK (1999)
6. Baufreton, P.: Visual notations based on synchronous languages for dynamic validation of GALS systems. In: CCCT'04 Computing, Communications and Control Technologies. Austin (Texas) (August 2004)
7. Bouhadiba, T., Maraninchi, F.: Contract-based coordination of hardware components for the development of embedded software. In: Proc. of COORDINATION. pp. 204–224. LNCS, Springer-Verlag, Berlin, Heidelberg (2009)
8. Chapiro, D.M.: Globally-asynchronous locally-synchronous systems. Ph.D. thesis, Stanford University (1984)

9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E., Sistla, A. (eds.) Computer Aided Verification, LNCS, vol. 1855, pp. 154–169. Springer Berlin / Heidelberg (2000)
10. Dajani-Brown, S., Cofer, D., Bouali, A.: Formal verification of an avionics sensor using SCADE. In: Lakhnech, Y., Yovine, S. (eds.) Proc. FORMATS/FTRTFT. LNCS, vol. 3253, pp. 5–20. Springer-Verlag (2004)
11. Doucet, F., Menarini, M., Krüger, I.H., Gupta, R., Talpin, J.P.: A verification approach for GALS integration of synchronous components. ENTCS 146, 105–131 (January 2006)
12. Garavel, H., Thivolle, D.: Verification of GALS systems by combining synchronous languages and process calculi. In: Proc. of SPIN. LNCS, vol. 5578, pp. 241–260 (2009)
13. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. LNCS, vol. 2102, pp. 53–65 (2001)
14. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. SIGSOFT Softw. Eng. Notes 28, 38–47 (September 2003)
15. Giese, H., Vilbig, A.: Separation of non-orthogonal concerns in software architecture and design. Software and System Modeling 5(2), 136–169 (2006)
16. Günther, H.: Bahnübergangsfallstudie: Verifikationsbericht. Tech. rep., Institut für Theoretische Informatik, Technische Universität Braunschweig (February 2012), Available at `http://www.versyko.de`
17. Günther, H., Hedayati, R., Löding, H., Milius, S., Möller, O., Peleska, J., Sulzmann, M., Zechner, A.: A framework for formal verification of systems of synchronous components. In: Proc. MBEES'12 (2012), available at `http:\\www.versyko.de`
18. Günther, H., Milius, S., Möller, O.: On the formal verification of systems of synchronous software components (extended version) (May 2012), Avaliable at `www.versyko.de`
19. Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous observers and the verification of reactive systems. In: Proc. of AMAST'93. pp. 83–96. Workshops in Computing, Springer-Verlag, London, UK (1994)
20. Halbwachs, N., Mandel, L.: Simulation and verification of asynchronous systems by means of a synchronous model. In: Proc. of IFIP. pp. 3–14. IEEE Computer Society, Washington, DC, USA (2006)
21. Halbwachs, N., Raymond, P.: Validation of synchronous reactive systems: from formal verification to automatic testing. In: Proc. of ASIAN (December 1999)
22. Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Proc. of EMSOFT. pp. 134–143. EMSOFT '07, ACM, New York, NY, USA (2007)
23. Jones, C.B.: Specification and design of (parallel) programs. In: Proc. IFIP Congress. pp. 321–332 (1983)
24. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Barnat, J., Heljanko, K. (eds.) PDMC. EPTCS, vol. 72, pp. 55–62 (2011)
25. Le Guernic, P., Talpin, J.P., Le Lann, J.L.: Polychrony for system design. Journal of Circuits, Systems and Computers (2002), special issue on Application-Specific Hardware Design. World Scientific
26. Milner, R.: Calculi for synchrony and asynchrony. Theoret. Comput. Sci. 25(3) (July 1983)
27. Möller, M.O.: Benchmark Analysis of GTL-Backends using Client-Server Mutex (2012), `http://www.verified.de/en/publications/`, Verified Systems International GmbH, Doc.Id.: Verified-WHITEPAPER-001-2012, Issue 1.2.
28. Mousavi, M.R., Le Guernic, P., Talpin, J., Shukla, S.K., Basten, T.: Modeling and validating globally asynchronous design in synchronous frameworks. In: Proc. of DATE. pp. 10384–. DATE '04, IEEE Computer Society, Washington, DC, USA (2004)
29. Rajan, B., Shyamasundar, R.: Multiclock Esterel: a reactive framework for asynchronous design. In: Proc. of IPDPS. pp. 201–209 (2000)

30. Ramesh, S.: Communicating reactive state machines: Design, model and implementation. In: Proc. IFAC Workshop on Distributed Computer Control Systems. Pergamon Press (September 1998)
31. Ramesh, S., Sonalkar, S., D'silva, V., Chandra R., N., Vijayalakshmi, B.: A toolset for modelling and verification of GALS systems. In: Alur, R., Peled, D. (eds.) CAV, LNCS, vol. 3114, pp. 385–387. Springer Berlin / Heidelberg (2004)
32. Sulzmann, M., Zechner, A., Hedayati, R.: Anforderungsdokument für die Fallstudie Bahnübergangssicherungsanlage. Tech. rep., ICS AG (2011)
33. Contract specification and domain specific modelling language for GALS systems, an approach to system validation. Tech. rep., ICS AG, Verified Systems International GmbH, TU Braunschweig (2011), Available at `www.versyko.de`