A framework for formal verification of systems of synchronous components

Henning Günther1, Ramin Hedayati2, Helge Loeding3, Stefan Milius1, Oliver Möller2, Jan Peleska2, Martin Sulzmann3, and Axel Zechner3

1Institut für Theoretische Informatik, Technische Universität Braunschweig
2Verified Systems International GmbH
3Informatik Consulting Systems AG

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 in a rely-guarantee style logic. Formal verification of global system properties is then done transforming a network of contracts to PROMELA/SPIN. Synchronous components are implemented in SCADE, and contract validation is done by transforming the contracts into synchronous observers and using the SCADE Design Verifier for formal verification. We also discuss first experiences from an ongoing industrial case study applying our approach.

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 SCADE1 is an attractive solution providing code generation and good support for (formal) verification and test automation. But for a distributed system, a synchronous implementation is neither realistic nor desirable. Hence, we focus on the model based 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 –

*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).
1SCADE is developed and distributed by Esterel Technologies: www.esterel-technologies.com
locally synchronous) architecture [Cha84], 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 for model checking verification goals that do not refer to real time. Later during the course of our work UPPAAL timed automata for real time verification goals will be used.

In the following we will describe our framework and the modeling languages for contracts and abstract GALS models (Sec. 2). Then in Sec. 3 we discuss our approach to system validation and give a brief description of the model transformations involved. In Sec. 4 we report about preliminary results using first prototypical implementations of the transformations on an industrial case study.

Related Work. Numerous publications are devoted to combining synchrony with asynchrony and the verification of GALS systems (see e.g. [GT09, DMK+06, RSD+04]) or, less related to our work, to use synchronous modeling towards less synchronous applications (see e.g. [RS00, PBC07], the tool Model Build [Bau99, Bau04], the Polychrony workbench [LGJPJL02] or the work in [HM06, JHR+07, MLGT+04]). 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 [Jon83]. Abstracting system components by contracts appears recently, for example, in [GV06, GTB+03, BM09]. Alur and Hinzinger [AH99] treat the semantics of asynchronous component systems.

Main Contribution. What is new in our work is the combination GALS verification with the idea of abstraction by contracts and its application to networks of synchronous SCADE models. 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 specification language for GALS systems as well as domain specific representations of it, and we design and implement model transformations between our language and appropriate model checking tools (PROMELA/SPIN and SCADE Design Verifier). A detailed description of the work presented in this paper can be found in the technical report [Ver11].

2 Modeling framework for GALS systems

In this section we describe our modeling framework for GALS systems. Its overall architecture consists of three layers as shown in Fig. 1. The top layer contains domain specific modeling languages for the end-user to specify GALS systems. We describe one such language, the Contract Domain Specific Language (CDSL) in Subsec. 2.1. The middle layer is formed by the GALS Translation Language (GTL). As described in Subsec. 2.2,
the GTL serves as a stable textual core language so that (1) on top of it one can define extensions and user friendly graphical representations tailored towards system engineers or certain application domains and (2) model transformations to test automation and formal verification engines are easy to define and may remain stable when languages at the top level are newly defined or changed. The bold arrows in the figure indicate model transformations and we describe some of them in Sec. 3. Dashed arrows indicate that each component in the abstract GALS model refers to a concrete synchronous (SCADE) implementation. Due to space constraints we do not describe the transformation of the CSDL to GTL, and the transformation pertaining to test case generation as well as the verification of GALS models using UPPAAL timed automata will be the topic of future work.

2.1 Domain specific contract modeling language for users

Elaborated as a profile of SysML, the CDSL is intended as a means for modeling SoS networks in an abstracted form at the top level of our framework in Fig. 1: nodes are represented by contracts abstracting their behavior in a way that still allows to verify system-level goals, but does not introduce the full behavioral information of the detailed model used to develop the node. The graphical high-level description is easier to comprehend than the (equivalent) GTL description, which is elaborated later. For the detailed CDSL description we refer to [Ver11].

A CDSL specification describes an inherently asynchronous network of synchronous components that are connected by interfaces. The network is represented by composite structure diagrams, its nodes by classes or composite structures, as will be explained below. While some nodes in the network may be unique and are therefore represented by sin-
gleton classes in the CDSL, many nodes may be regarded as multiple instances of the same class. These instances are distinguished by certain parameters (in the simplest case, numerical identifiers) which are specified as class attributes and can be defined via constructor invocation during the instantiation process. Since SoS usually consist of a very large number of nodes, the CDSL model abstraction does not require to draw object diagrams for representation of the SoS network; instead, the composite structures and classes are associated with instantiation rules in a tabular format, showing how many objects are created from their respective classes. This concept is applied to both behavioral objects representing nodes or node components, and interface objects.

A contract consists of an interface description (represented by the <<interface>> stereotype defined in UML2) associated with a class or, for more complex nodes, a composite structure. Classes or composite structures specify the behavior of the node in abstracted form. The class, decorated with a contract stereotype which is introduced by the CDSL profile, offers two methods rely(c:computation):boolean and guarantee(c:computation):boolean whose bodies specify the rely and the guarantee parts of the node’s contract. For these specifications (timed) LTL formulas ρ (rely) and γ (guarantee) are used. Their free variables are the attributes defined in the node’s interface. The bodies of the two methods are defined by return (c |\= ρ); and return (c |\= γ);, respectively.

Alternatively the class can be associated with a timed state machine or a decision diagram acting on the interface attributes and, optionally, auxiliary local variables specified as class attributes. This specification style expresses the rely-part of the contract in an implicit way as the set of all possible input sequences of the state machine. The body of the rely() method is now associated with constraint (c ∈ statemachineinputs). The guarantee-part is represented by the state machine’s traces, projected to the input and output attributes of the interface. The body of the guarantee() method is specified as return (c ∈ state machine computations);

In addition to the rely/guarantee assertions the contract may specify guaranteed behavior, modeled by a method guaranteed(c:computation) : boolean which is also associated with an LTL formula γ’: 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 system-level verification to uncover flaws in contract specifications or verification goals.

More complex nodes may be represented by a hierarchy of composite structure diagrams, with contract classes in their leaf nodes, in order to structure the component’s behavior into sub-component behaviors. The sub-components act synchronously, i.e., run-to-completion steps do not consume time. Data flow between arbitrary elements is captured again by the standard interface concept, where each interface transports one or more typed values.

Example. Component GEN in Fig. 2 generates a pair of values (a, b), which is used (asynchronously) by the composite structure PROC to compute RESULT = (a + b)^2. Components ADAPT, and MUL are (synchronous) sub-components of PROC, each instantiated four times. The special parameter id identifies the instance, both for components and for interfaces. ADAPT[id] organizes the input by means of interfaces X[id] and Y[id].
such that the four multiplications to be processed by $\text{MUL}[0]$ to $\text{MUL}[3]$ are $a \times a$, $a \times b$, $b \times a$, and $b \times b$, respectively. The one instance of $\text{SUM}$ sums up these intermediate results, which yields finally $\text{RESULT} = a^2 + 2ab + b^2 = (a + b)^2$. If, for example, $\text{GEN}[0]$ provides a guarantee $\gamma$: “always $a + b = 1$”, then system-level verification allows to infer the system property “always $\text{RESULT} = 1$”.

2.2 Textual contract specification language GTL

The GTL is used to specify networks of synchronous components and is intended as a simple textual language serving as a source for model transformations to model checkers and test automation tools. Fig. 3 shows a simplified and abbreviated GTL-specification from the industrial case study in Sec. 4. Each synchronous component is an instance of an abstract model. A model is declared by a name, the synchronous formalism in which it is implemented and formalism-specific information on how to retrieve the synchronous model and its interface description (e.g., the name of the model file), see lines 1 and 8. Inside the curly braces, the user can specify the cycle time of the synchronous model and its contract. Contracts are specified by one or more LTL-formulas over the input and output variables of the component (line 4). For example, the rely and guarantee parts of a contract in CDSL yield the formula $\rho \Rightarrow \gamma$. In addition, it is also possible to specify contracts by using state machines, which can use local variables to describe the behavior of the model (line 11-21). Models may also specify guaranteed behavior; this is done by LTL formulas following the keyword guaranteed in the model’s body. Components can be declared by creating instances of models, as displayed in line 24-26. Instances may extend the contract of their model by LTL formulas, automata and/or guaranteed behavior. All declared instances can then be connected to a network by connecting input and output variables together (line 27). LTL formulas can be used to specify verification goals. Those formulas can use all in- and output variables of any component in the system (lines 30). It is usually unclear what the successor of a state of a GALs model is, so verification goals will use temporal connectives: the formula specifies that an event must happen no later than two seconds after another event.
3 Validation of GALS systems

In this section we describe how GALS systems are formally verified in our framework. We first explain the general approach to system level verification, and then we outline how various model transformation from the GTL to different analysis tools are used to realize formal verification.

3.1 System-Level Verification Approach

As indicated above, our approach advocates system-level verification along the following lines: (1) System-level verification goals $\Phi$ are specified as (timed) LTL formulas expressing the desired behavior of the complete GALS system. (2) The behavior of each synchronous component $C$ is abstracted by its contract $\Phi_C$ which may be represented by LTL formulas or, alternatively, abstracted models, and, optionally, guaranteed behavior represented again by LTL formulas $\beta_C$. (3) From the network of contracts an abstract
GALS model $M_G$ is derived, represented by a network of most non-deterministic component processes $C'$ still satisfying the contracts $\Phi_C$. (4) The assertion "system satisfies $\Phi$" is verified by property checking $M_G \models \Phi$.

The system-level verification result has to be validated with respect to the following verification threats: (VTH-1) the verification failed because the system is inadequate for its purpose, that is, $\Phi$ correctly reflects a system requirement, but this is not fulfilled by the designed system. (VTH-2) The verification fails though the system is adequate for its intended purpose (so-called false negative). (VTH-3) The verification succeeds though the system is inadequate for its intended purpose (false positive).

(VTH-1) represents a “desired” verification result: the proof of $M_G \models \Phi$ failed because some or more components $C$ have been inadequately modeled, and this inadequacy has been duly reflected in their contracts $\Phi_C$.

(VTH-2) may have 3 root causes: firstly, the verification goal $\Phi$ may have been inadequately specified. This situation is most easily uncovered by analyzing the error trace $\pi$ produced by the model checker as a witness for the violation of $M_G \models \Phi$, and determining whether $\pi$ really reflects undesired system behavior. Secondly, a contract $\Phi_C$ may have been specified too weakly, so that the projection $\pi_C$ of $\pi$ to the interfaces of $C$ results in a computation which is not a computation of $C$’s detailed model. This can be discovered by running the projection against a simulation generated from the detailed model of $C$. As a consequence $\Phi_C$ has to be strengthened. Instead of elaborating a completely revised version of $\Phi_C$ it may be helpful to investigate whether $\pi_C$ is inconsistent with the guaranteed behavior $\beta_C$ of $C$, in symbols: $\pi_C \not\preceq \beta_C$. In this situation, it suffices to strengthen the contract by adding $\beta_C$ as a conjunct to the original version of $\Phi_C$. Thirdly, the contract may be inconsistent to $C$. This case should have been already detected during contract elaboration, by means of property checking $M_C \models \Phi_C$.

(VTH-3) reflects the situation where at least one component $C$ is faulty, but this is not detected in the verification of $M_G \models \Phi$. This may be caused by a specification error in $\Phi$ or by an undetected contract inconsistency masking the faulty behavior of $C$. The latter case is handled as described for (VTH-2). The former may be uncovered if simulation traces $\pi$ are generated by exercising model-in-the-loop testing techniques on $M_G$: if a test fails but the associated simulation trace is accepted by $\pi \models \Phi$, the inadequacy of $\Phi$ is revealed. This technique relies on redundant specification of required behavior: instead of just using $\Phi$ as a global test oracle, tests are associated with their own specifications of expected results. Again, the guaranteed behavior specifications $\beta_C$ may be applied to strengthen these expected test results.

3.2 Realization by Model transformations

In this section we explain how various analysis tools are used for the validation of GALS systems described in the previous section. This is achieved by using model transformations as indicated by the bold arrows in Fig. 1. Due to space constraints we only describe the translation principles of some translations very briefly, a more detailed account

---

2Observe that the abstracted GALS model $M_G$ operates on the complete concrete interfaces specified for each component $C$, so that abstraction only introduces more general behavior, but not abstracted data.
is in [Ver11].

**Local verification of contracts.** Contract inconsistencies can be uncovered using local model checking. For a component $C$ implemented by a SCADE model one transforms the contract $\Phi_C$ into a so-called synchronous observer node in SCADE (cf. [HLR94, HR99, DBCB04]). Each LTL formula contained in the contract is first translated to a state machine using the translation algorithm described by Gastin and Oddoux [GO01]. As a result, for each component $C$ we obtain a set of automata. Then the product automaton is formed to obtain a single automaton describing the behavior specified by $\Phi_C$. This automaton is transformed into a SCADE synchronous state machine (cf. [And03]) representing the observer node, and SCADE Design Verifier is used to verify whether $M_C \models \Phi_C$. Notice that due to restrictions of the Design Verifier, it is only possible to verify contracts with bounded liveness properties.

**Checking verification goals.** In order to verify whether an abstract GALS model $M_G$ satisfies a verification goal $\Phi$ the network of contracts is transformed into a PROMELA model. Each contract is transformed into an automaton as described above, and for each contract automaton a PROMELA process is created. The different processes asynchronously communicate via shared variables that correspond to the connections of input 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.) In addition, our model transformation creates a scheduler that facilitates the fair synchronous execution of the components: in the PROMELA model all component start at the same time and then proceed according to their cycle times. If the verification goal $\Phi$ is an ordinary LTL formula it can simply be verified whether $M_G \models \Phi$ by using SPIN. If $\Phi$ contains temporal connectives, timers are introduced in the PROMELA model. For example, the formula $\phi \text{ until } [t] \psi$ holds in a state $s$ of $M_G$ if $M_G, s \models \phi \text{ until } \psi$ and for every run $s = s_0, s_1, s_2, \ldots$ of $M_G$ from $s$ a state $s_i$ with $M_G, s_i \models \psi$ is reached within time $t$. The formula is translated to

\[(c := t) \land ((\phi \land c \geq 0) \text{ until } (\psi \land c \geq 0));\]

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.

**Detecting false negatives.** The third transformation from GTL depicted in Fig. 1 can be used to validate verification results for an abstract GALS model. Suppose we have $M_G \not\models \Phi$ for a verification goal $\Phi$ and the formal verification produces the counterexample trace $\pi$. If each component comes with a SCADE implementation, we can check whether this is a real error trace or a false negative as follows: from the GTL specification one generates a concrete GALS model $M'_G$ which composes the SCADE models of the components by integrating the C-code generated from them. By using SPIN to simulate $M'_G$ on the inputs from $\pi$ we can verify whether $\pi$ is indeed an error trace showing $M'_G \not\models \Phi$, or equivalently, $\pi$ is a legal trace of $M'_G$.

---

3Again, this simulation is possible since $M_G$ operates on the complete concrete interfaces specified by each component $C$ (cf. e.g. Fig. 3 line 1)
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 error trace $\pi$ on a local trace $\pi_C$ for each component $C$, which can be used in the ensuing analysis: from each $\pi_C$ one can generate a SCADE simulator script which can be used to correct the SCADE models of the components. If the simulation finds that $\pi$ 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.

4 Case studies

We have successfully tested our approach and the prototypical tools on various small academic examples. We are also developing two real industrial case studies to demonstrate the use of our approach to GALS system verification: the first case study is a cabin smoke detection system of an airplane and the second one is a level crossing system from the railway domain. Due to space constraints we only describe the latter case study here, details concerning the former can be found in [Ver11].

The level crossing in the case study consists of several modules (traffic lights, supervision signals, barriers etc.). An overview of the level crossing architecture is given in Fig. 4. The modules have been implemented as synchronous reactive components of medium complexity within SCADE. The implementation can be found at the VerSyKo project web page. The level crossing fulfills the main requirement to protect the road traffic from the train traffic and vice versa. More detailed user- and system-requirements, on which the implementation is based, can be found in [SZH11]; this provides a more detailed informal description of the level crossing system and the overall system architecture.

The user- and system-requirements from [SZH11] need to be verified outside the SCADE environment since the level crossing system has a GALS architecture. In a first experiment we have integrated the C-code generated from some of the SCADE models of the components and used the model checker SPIN for global verification. As expected, this yielded no result due to state space explosion, even using a reduced system model, validating our expectation that it is necessary to reduce state space by providing abstractions of the local synchronous components using contracts.

For the level crossing case study we have formulated contracts for each of the components of the level crossing system. This experience shows that it can be non-trivial to define a
correct and adequate abstraction that is qualified for model checking, and leads to diagnostically conclusive result. It may be necessary to deeply investigate the implementation, and there seems to be no simple automated solution for deriving contracts. However, for the traffic light controller the contract validation has revealed a subtle error in the implementation. For two states in the $\text{SCADE}$ 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. Unfortunately, after this correction $\text{SCADE DV}$ did not succeed to verify our contract correct; we only managed to prove correctness for a simplified version of the component’s $\text{SCADE}$ model.

First experiments with global verification show limitations of our approach to verification of GALS systems in connection with explicit state model checking as in $\text{SPIN}$. To address these problems we currently investigate a different approach using bounded model checking and a model transformation for global verification from $\text{GTL}$ to an $\text{SMT}$-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.

References


