Automatic Generation of Assumptions for Modular Verification of Software Specifications

Claudio de la Riva and Javier Tuya

Computer Science Department - University of Oviedo
Campus of Viesques, E-33207, Gijón (SPAIN)
Phone: +34 985 182664
Fax: +34 985 182156
[claudio|tuya]@uniovi.es

Abstract. Model checking is a powerful automated technique mainly used for the verification of properties of reactive systems. In practice, model checkers are limited due to the state explosion problem. Modular verification based on the assume-guarantee paradigm mitigates this problem using a "divide and conquer" technique. Unfortunately, this approach is not automated, for the reason that the user must specify the environment model. In this paper, a novel technique is presented for automatically generating component assumptions based on the behaviour of the environment (the remainder of components of the systems). In a first phase, the environment of the component is computed using state space exploration techniques, and then the assumptions are generated as association rules of the component environment interface. This approach presents a number of advantages. Firstly, user assistance to specify assumptions is not necessary and assumption discharge is avoided. Secondly, the component assumptions are more restrictive and real, and therefore reduce the resources needed by the model checker. The technique is applied to the specification of a steam boiler system.

Keywords: assume-guarantee reasoning, model checking, modular verification, component verification.

1 Introduction

Model checkers are powerful and useful automated tools for the verification of finite state systems (Clarke et al., 2000). Given a property and a model of the system, the model checker performs an exhaustive state space exploration of the system and returns true if the property holds in the system or a *counterexample* (a scenario illustrating the violation of the property) otherwise. These techniques have traditionally been used in the verification of hardware devices. Recently, however, many researchers have started to apply model-checking techniques to software systems, ranging from specifications (Sreemani and Atlee, 1996; Chan et al., 1998) to source code (Chaki et al., 2004). However, the main drawback of these tools is the *state space explosion* problem: in general, the size of the state space that the model checker must handle grows exponentially with the number of processes and states that constitute it. This is the case of *reactive* systems, since they are composed of many parallel, concurrent processes and they maintain continuous interaction with the environment. By introducing techniques such as partial order reductions (Peled, 1994) and symbolic representation of states and transitions (McMillan, 1993) into model checking procedures, systems with large state spaces can be verified. However, these techniques still present limitations, and many realistic systems are not tractable.

A promising natural approach to avoid the state explosion problem is to use "divide and conquer" techniques: the system is broken down into components and each of these is verified separately from the rest. Thus, the problem of verification of the global system is downsized to the analysis of its individual components (*compositional* or *modular* verification). However, there are several issues that make using modular verification difficult. First, deciding how to partition the system is not trivial and can have a significant impact on the resources needed for verification (Cobleigh et al, 2004). Second, when model checking a component in isolation, a model of the environment interacting with the component must be introduced. If this environment it is not supplied, the "*universal environment*" or the "*most general*

Final preprint version published in Journal of Systems and Software, Vol. 79(9) pp. 1324-1340.
© Elsevier, doi:10.1016/j.jss.2005.11.570

environment" is considered: i.e. the component can receive and send any event and in any order from the environment. Obviously, this approach is not real because components are designed to operate in a concrete environment. This approach is considered *pessimistic* (Alur et al., 1998; Giannakopoulou et al., 2005). McMillan (McMillan, 2000) calls this problem the *environment problem*.

This paper focuses on the second problem employing an *assume-guarantee* framework (Jones, 1983; Pnueli, 1984; Abadi and Lamport, 1995). This style of reasoning addresses the environment problem by specifying the environment using *assumptions* that are manually provided by the engineer. The property is first verified on the component with the assumptions, and then the environment must *guarantee* these assumptions, i.e. they must be verified on the remaining components of the system (the environment).

Although modular verification based on assume-guarantee rules has been extensively studied, its application to large-scale problems is scarce (Shankar, 1997) due to the fact that it partly needs user guidance. In general, the assumptions are incorporated into the component based on user knowledge and the feedback obtained from the counterexamples (Giannakopoulou et al., 2005). Firstly, the component is checked without assumptions (with the universal environment). If the model checker returns an unrealistic counterexample or *false positive*, the user makes a number of attempts to define a set of appropriate assumptions that eliminate the false positive and which in turn reflect an adequate behaviour of the environment.

This approach has various weaknesses. Firstly, although verification using the model checker is fully automatic, modular verification using the aforementioned process is not. The engineer must provide the assumptions, which is a difficult task, and moreover, these assumptions must be discharged onto the environment. Secondly, the process is error-prone because manual assumptions cannot sufficiently restrict the environment and thus produce false positives. In short, the approach is time-consuming in human effort and computational resources because of the manual specification of assumptions and their discharge onto the environment.

In this paper, we develop a novel technique for providing automated support to modular verification based on the assume-guarantee reasoning of software specifications. Given a component *P* to verify, the state space of the environment is computed in the *Composition* step and the set of all possible configurations of the environment *E* is obtained. In a second step (*Extraction*), event relations between the component and its environment (assumptions) are computed using *association rules*. The component is then verified along with the assumptions obtained by a model checker. By applying this process to each component of the system, the system can be verified in a modular automatic fashion, substantially reducing the state space explosion and side-steeping user guidance (the user will only be required to partition the systems into components and local properties). Fig. 1 illustrates the approach.

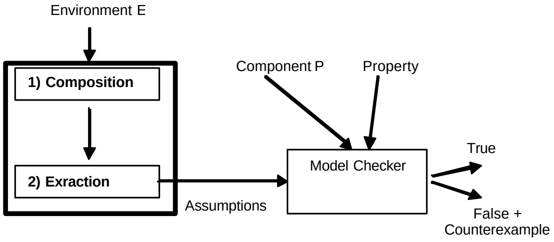


Fig. 1: Assumption Generation.

The technique is applied to software specifications modeled as state transition systems synchronized by events whose semantics are based on two time levels: external interactions or *macrosteps* and internal interactions or *microsteps* (Harel et al., 1987; Leveson et al., 1994).

The rest of the article is organized as follows. We present the underlying computational model in Section 2. The Composition and Extraction steps of our approach are presented in detail in Sections 3 and 4, respectively. Section 5 is given over to showing the results obtained with the application of the method to various components in a steam boiler control program. These results and other topics relative to the applicability of the approach are discussed in Section 6. Related work is outlined in Section 7 and we present our conclusions in Section 8.

2 Background

In this section, we describe the syntactic and semantic framework of the system on which our approach is based and review the foundations of modular verification based on assume-guarantee reasoning.

We use finite state machines (FSMs) to model the individual processes of the system. An FSM contains the states that a process may reach and the transitions that it may perform. Formally, an FSM is a 5-tuple *(S,s$_o$,I,O,d)*, where *S* is the set of states, $s_o \in S$ is the initial state, *I* and *O* are the set of input and output events of the process, respectively, and *d Í S ´I´O´S* is the transition relation between states of the FSM. A transition *t=(s,c,a,s')Î d* (s®$_a^c$s') means that the process executes the transition when it receives the input event *c*, changing to state *s'* and producing the set of events *a*⊂*O*. Each FSM implicitly has a *control state p*, which represents the local state of the process (initially *p*=*s$_o$*). A transition *t=(s,c,a,s')* is *enabled* if *p*=*s* and the evaluation of input events condition *c* is true. The set of enabled transitions in a state *s* is denoted by *enabled*(*s*).

2.1 Composition of processes: synchronous reactive systems

An FSM captures the basic behaviour of a process. However, a component in a reactive system is formed by a set of processes interacting in parallel. We will use *Synchronous Reactive Systems* (*SRS*) to model the behaviour of communicating components in a concurrent, reactive system. Intuitively, an SRS is a set of FSMs interacting with the environment over a set of *external* input and output events. The events that internally communicate FSMs are named *internal* events. Fig. 2 shows a simple system modelled as an SRS composed of two processes. *P1* receives two external input events from the environment, *INIT* and *FINISH*, and sends the internal events *Start* and *Stop* to *P2*. Depending on its local state, *P2* produces the external output events *OPEN* and *CLOSE*.

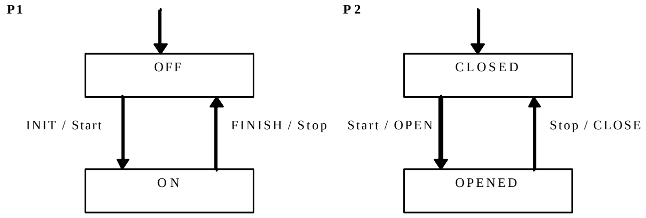


Fig. 2: An SRS example

To describe the behaviour of an SRS, we adopt the step-based semantics described in the RSML language (Leveson et al., 1994) and in the statecharts tool STATEMATE in asynchronous mode (Harel and Naamad, 1996). A *macrostep* starts when external input events arrive. Internally, a *macrostep* is formed by a chain of *microsteps*. At each of these microsteps, the system reacts to the input events executing locally enabled transitions in the FSMs and producing external output events and internal events that may initiate another microstep. The macrostep is completed when no more transitions can be executed. At this point, the output events generated during the macrostep are sent to the environment at the same time. Fig. 3 depicts these concepts.

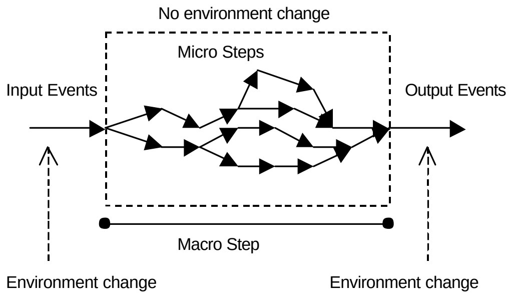


Fig. 3: Macrosteps and microsteps

For example, in the SRS in Fig. 2, supposing that both processes *P1* and *P2* are in their initial states, the environment produces the *INIT* external input event that starts a macrostep. The process *P1* executes the local transition from *OFF* to *ON* and generates the internal event *Start* (first microstep), which initiates a second microstep executing the transition from *CLOSED* to *OPENED* in *P2*. No more transitions can be executed in this configuration and the macrostep finishes, sending the (external) output event *OPEN* to the environment.

More specifically, an SRS Φ is a 5-tuple (*D,GE,IE,OE,®*^{m}) where Δ is the set of FSMs, *GE* is the set of internal events that communicate each FSM, *IE* and *OE* are the external input and output events, respectively, and $\rightarrow^m$ is the transition relation that describes the semantics outlined above and which we will define below. The initial state of an SRS is made up of the set of initial states of the FSMs. A (global) state is composed of the control states of the FSMs. We denote by *C*=(*S,G,I,O*) a *configuration* of the SRS, where *S* is the state of the SRS and *G*, *I* and *O* are the values of internal, external input and external output events of the system, respectively. The set of all possible configurations of Φ is denoted by *Global*(Φ). We provide the semantics for Φ on the basis of the definition of the transition relation ®^{m}⊆*Global(*Φ)×*Global(*Φ), using the following inference rules.

Start Rule: This represents the commencement of the macrostep and the first microstep execution. The environment changes and new external input events enter the system. It applies to each FSM Mi if it has an enabled transition τ in a state si of the initial configuration of the macrostep. If Mi has multiple enabled transitions in its configuration, one of them is taken non-deterministically. If various FSMs have enabled transitions in the initial configuration of the macrostep, then each of them is executed simultaneously:

$$\frac{(p_i = s_i) \wedge (\exists t : s_i \rightarrow^{a_i}_{c_i} s'_i \in d_i / t \in enabled(s_i))}{((p_1,\ldots,s_i,\ldots,p_n),\varnothing,I,\varnothing) \rightarrow^m ((p_1,\ldots,s'_i,\ldots,p_n),G^1,\varnothing,O^1))}$$

where G^1={e∈a$_i$/e∈GE} and O^1={e∈a$_i$/e∈OE} are the internal and external output events, respectively, generated as a consequence of the first microstep execution.

Advance Rule: This represents the execution of a microstep *k* (*k*=1) as a consequence of the internal events G^{k-1}∈*GE* generated in the previous microstep. As in the start rule, if various FSMs have enabled transitions, each of these is executed simultaneously:

$$\frac{(p_i = s_i) \wedge (\exists t : s_i \rightarrow^{a_i}_{c_i} s'_i \in d_i / t \in enabled(s_i))}{((p_1,\ldots,s_i,\ldots,p_n),G^{k-1},\varnothing,O^{k-1}) \rightarrow^m ((p_1,\ldots,s'_i,\ldots,p_n),G^k,\varnothing,O^{k-1} \cup O^k))}$$

Stuttering Rule: If an FSM M_i is a state in the current configuration of the microstep and it does not have any enabled transitions, then it consumes events but it does not produce events:

$$\frac{(p_i = s_i) \wedge (\forall t : s_i \rightarrow^{a_i}_{c_i} s'_i \in d_i / t \notin enabled(s_i))}{((p_1,\ldots,s_i,\ldots,p_n),G^{k-1},\varnothing,O^{k-1}) \rightarrow^m ((p_1,\ldots,s'_i,\ldots,p_n),\varnothing,\varnothing,O^{k-1}))}$$

Stop Rule: This represents the end of the macrostep. Whereas the above rules apply to each FSM and show the SRS execution at a microstep level, this rule applies to the SRS and defines when the macrostep concludes. When every process can only execute the stuttering rule, the SRS has reached a *stable* *configuration* and the macrostep ends, sending to the environment the output events $O^1 \cup O^2 \cup \ldots \cup O^n$ produced in the chain of microsteps.

2.2 Synchronous r eactive components

In order to perform modular verification, the global system must be broken down into components so that each one may be verified separately. From this point of view, a component is formed by a set of FSMs and is modeled as an SRS. However, when we perform local verifications in the components, internal events that communicate them must be treated differently from the rest because they can take any value in each microstep. We define a *Synchronous Reactive Component* (or *component* for short) as an SRS in which the set of internal events are divided into internal events of the component $\Gamma \subset GE$, input interface events $\Psi \subset GE$ (internal events produced by other components) and output interface events $\Omega \subset GE$ (internal events used by other components). We denote by *C=(S,G,IntI,IntO,I,O)* the configuration of the component, where *G*, *IntI* and *IntO* are the values of the internal (Γ), the input (Ψ) and output interface events (Ω), respectively. Since a microstep can start when an input interface event reaches the component, the advance rule of an SRS must be slightly modified here to define the execution of a microstep in a component:

$$\frac{(p_i = s_i) \wedge (\exists t : s_i \xrightarrow[c_i]{a_i} s'_i \in d_i \,/\, t \in enabled(s_i))}{((p_1,\ldots, s_i,\ldots,p_n), G^{k-1}, IntI^{k-1}, IntO^{k-1}, \varnothing, O^{k-1}) \rightarrow^m ((p_1,\ldots, s'_i,\ldots,p_n), G^k, IntI^k, IntO^k, \varnothing, O^{k-1} \cup O^k)}$$

The rule states that a microstep k can be activated by either the values of the internal events G^{k-1} generated in the preceding microstep or the values of the input interface events $IntI^{k-1}$ received in the preceding microstep. The remaining rules of the SRS are the same and can be directly translated to the component.

2.3 Assume-guarantee modular verification

As mentioned in the introduction, the main problem of automatic tools (i.e. model checkers) in the verification of realistic system is the state space explosion problem. A natural approach to solving this problem is to divide the system into components and to perform local verifications on separate components and then to deduce a number of global properties for the whole system (*compositional* or *modular verification*) (de Roever et al., 2001).

Assume-guarantee reasoning handles modular verification using environment component assumptions that the environment must subsequently guarantee. In this paper, we use Abadi and Lamport's compositional theorem (Abadi and Lamport, 1995): if *P* and *Q* are the components of the global system *S*, we can perform the verification of local properties j_P and j_Q for each component assuming some kind of behaviour for the other components. For instance, we can prove that j_Q is true for component *Q* assuming a behaviour ϕ_P of the component *P*. Next, we must verify that the behaviour ϕ_P assumed for *P* is true (*assumption discharge* or *commitment*). If this process is applied in this way to property j_P, we may conclude that (j_PÙ j_Q) is true for the whole system *S*.

In the context of the above computational model, the assumptions specified for a component are related to its input interface. For instance, suppose the system *S* is formed by two components *P* and *Q* and their respective input interface events are $\Psi_P=\{e_1, e_2\}$ and $\Psi_Q=\{e_3\}$. The assumptions for the component *P* will be formulated in terms of the events in Ψ_P and for the component *Q* in terms of the event in Ψ_Q, and will be specified using a first order logic formula. For example, the assumption "*when* e_1 *occurs,* e_2 *does not occur*" relative to the component *P* can be encoded as $(!(e_1=1) \wedge (e_2=0))$.

3 Composition step

The goal of this stage is to obtain the set of *all* the configurations of the environment to be used in the next step for generating assumptions (Section 4).

Let S=[E||P] be the whole system, where P is the component to verify and E is the component that represents its environment. This step computes the set *Global'*(E)=*Global(E)*↑($\Omega_E \cap \Psi_P$), where the operator ↑ erases from *Global*(E) those references to output interface events of the environment E (Ω_E) that are not part of the input interface with component P (Ψ_P). Using a reachability *depth-first search* algorithm and starting out from the initial configuration of the component E with each possible value in the input events, new configurations are generated using the microstep operator $®^m$ rules defined in Section 2.1 and Section 2.2. The process continues with each new reachable state and concludes when all the configurations are stable (no more microsteps can be executed from reachable states). Because the microstep operator ($®^m$) is applied to each new microstep configuration generated, the approach contemplates non-deterministic situations, guaranteeing that all configurations are explored.

Fig. 4 summarizes this approach. The shaded nodes represent stable configurations (reachable states) and the rest are the configurations corresponding to a microstep. The arrows from a stable configuration represent the execution of a microstep with a combination of input event values using the start rule. The arrows that connect configurations represent the execution of microsteps using the start or advance rule, and the dashed arrows execution using the stuttering rule. The path between two stable configurations defines a macrostep. The graph is computed from top to bottom and then from left to right, so that the different macrosteps are constructed in their entirety. Fig. 5 shows an example for the component in Fig. 2.

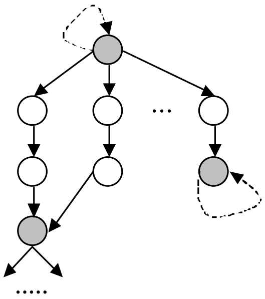


Fig.4: General Strategy to Compute Configurations

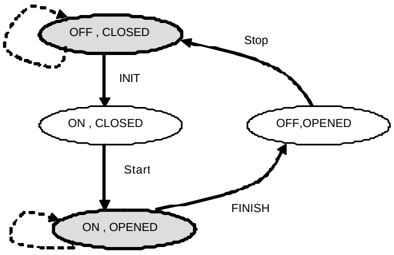


Fig. 5: Sample Reachability Graph for Fig. 2

3.1 Open systems

The previous basic composition step considers the environment to be a *closed* component: the macrosteps are constructed on the basis of the set of values of the input events by means of a microstep sequence (Fig. 4). In a compositional framework, however, the components must be considered as *open* systems that interact with other components and with the (physical) environment. Hence, the reachability graph for each component of the system may be independently derived from the complete system (Graf et al., 1996; Cheung and Kramer, 1996). In terms of the semantics defined for a component (Section 2.2), this means that events from other components may arrive at any microstep due to the fact that in the global system these events are internal events. Thus, we do not alter the semantics of the complete system when we consider a system component in isolation.

To illustrate how these observations affect the process of configuration generation, consider once more the component in Fig. 2 (with processes P1 and P2), subsequently modified in Fig. 6 with an input interface event *Failure* from another component (not shown in the figure) to process P2 and its reachability graph (Fig. 7). In order to compute all the configurations of the component, we must take into account the fact that the event *Failure* may be present at any microstep. This produces new microstep configurations, as shown in the graph: a microstep corresponding to the execution of transition from the state <*ON,OPENED*> to <*ON,CLOSED*>, another from <*OFF,OPENED*> to <*OFF,CLOSED*> and the stuttering transitions in each state.

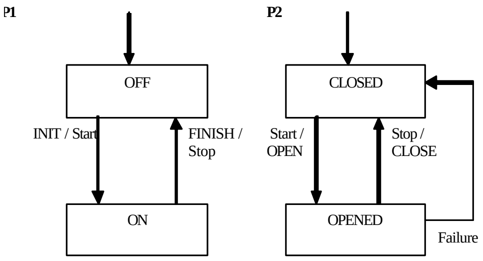


Fig. 6: An *open* component example with two processes

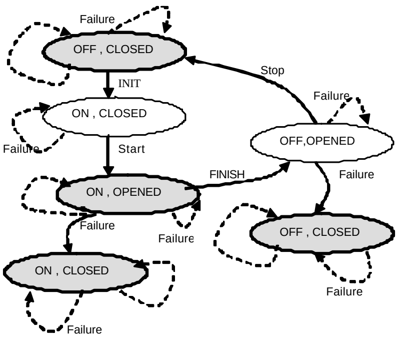


Fig. 7: Reachability graph for Fig. 6

The computation of configurations in open systems increments by one the length of all macrosteps and a new microstep is needed to decide whether it enables a transition. For example, consider the microstep configuration <*OFF,CLOSED*> corresponding to the ending of the macrostep in Fig. 7. Implicitly, a microstep exits from <*OFF,CLOSED*> to itself to check whether new transitions are enabled as a consequence of the appearance of the event *Failure* in the previous microstep. The new microstep is denominated a *stabilization microstep* because the transition is not executed, but rather is used to control the macrostep completion.

3.2 Microstep counter

In order to be able to reason about the instant at which the environment reacts and use this information in the assumptions, we made the microsteps visible. The aim is to make it possible to generate assumptions in the Extraction step about concrete and explicit microsteps for all macrosteps: for example, "*in microstep 1, the event OPEN is always 0*". This explicit reference to the microsteps will enable us to obtain a more detailed and real view of the behaviour of the environment and hence a better performance of the model checker because it limits the values of the interface events.

We include a new variable *mS* associated with each configuration that performs the function of a *microstep counter*: it is incremented in each microstep and reset at the beginning of a new macrostep (stable configuration) and the initial configuration. That is, if *C* and *C*' are the configurations at the beginning and end of the microstep, respectively, we have $(C,mS)\rightarrow^{m}(C',mS+1)$. For example, in Fig. 7 the configuration related to the transition from <*OFF,CLOSED*> to <*ON,CLOSED*> as a consequence of the appearance of external input event *INIT* is the following:

(<*ON*;*CLOSED*>, *Start*=1;*Stop*=0 ,*Failure*=0; ∅, *INIT=0*;*FINISH*=0, *OPEN*=0;*CLOSE*=0, *mS*=1)

The configuration reflects the first microstep execution by means of the microstep counter (*mS*=1) and the internal event *Start*=1. The rest of events take value 0.

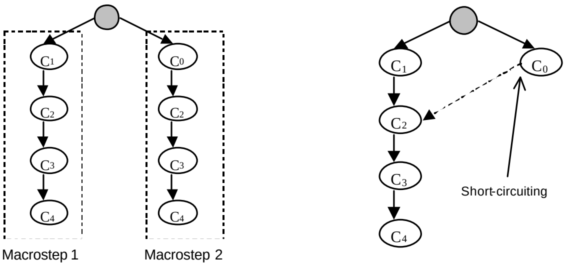
Since we think about the components as open systems, so as to be able to reason about the environment, compilation of the synchronous component cannot be carried out prior to its analysis in order to eliminate the microsteps and hence obtain a more compact state space (as in (Jajadeesan et al., 1996)). This explicit representation of all microsteps allows us to state the adequate assumptions and prove them without any distortion of the semantics of the global model.

Microstep visibility has been used in (Chan et al., 2001), though for different purposes that are not directly applicable to our approach. These authors use a microstep counter to optimise the analysis of statechart-like specifications, making every macrostep equal in length to prune backward searches in BBD-based model checking.

3.3 Microstep short-circuiting

Microstep visibility (the microstep counter) and the environment model as an open system could produce a similar effect to the state space explosion, which we call the *configuration explosion*.

To alleviate this potential problem, we take into account the following consideration. Given that the state space of configurations is computed sequentially following a depth-first search strategy for each reachable state, if a microstep configuration has already computed, the generation of the remaining chain of the microstep is interrupted from this configuration (*microstep short-circuiting*), due to the remainder of the microstep being equal to that computed in some previous macrostep. Fig. 8 illustrates this idea, first with two macrosteps without microstep short-circuiting and then with the resulting macrosteps after microstep short-circuiting optimisation. This substantially reduces the size of the state space of configurations. Obviously, the effect may be greater when the system executes a higher number of microsteps.



Macrostep 1: $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$ Macrostep 1: $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$
Macrostep 2 : $C_0 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$ Macrostep 2 : $C_0 \rightarrow C_2$

Fig. 8: Calculated macrostep without (left) and with (right) microstep short-circuiting

The decision as to when to apply microstep short-circuiting is made in each microstep by comparing the generated microstep configuration and the previously computed configurations. To improve the configuration comparison process, we compare enabled transitions in a microstep configuration with a table indexed by the microstep counter, where each record stores the set of enabled transitions per microstep. Thus, when the enabled transitions are computed for a microstep, these are compared with the transitions stored in the record corresponding to the microstep. If both sets are equal, the microstep is short-circuited. Because all transitions are compared, and hence explored, all possible different configurations are also computed. In consequence, the microstep short-circuiting does not affect the non-determinism.

4 Extraction step

The goal of this step is build the assumptions from the sets of configurations of a component computed in the previous stage. To do so, we will compute the assumptions as a set of *association rules*, adapting the *Apriori* algorithm (Agrawal and Srikant, 1994) used in the discovering of association rules. We will briefly describe this algorithm, to then go on to spell out the adaptations made for our purposes.

4.1 Association rules and the *Apriori* algorithm

The problem of discovering *association rules* between attributes in databases (in general, sets of transactions) was introduced in (Agrawal et al, 1993) and is a typical problem in knowledge discovery in databases. Given a set of transactions, in which each transaction is a set of items, an association rule is an implication with the form $X \Rightarrow Y$, where X and Y are sets of items. Two notions for establishing the strength of a rule are defined: the *confidence* of a rule is the fraction of transactions containing X that also contain Y, the support is the fraction of transactions that contain both X and Y. The goal is to find all the associations from large sets of transactions that satisfy user-specified minimum support (*minsup*) and user-specified minimum confidence (*minconf*). The initial idea is based on an itemset approach and is as follows. In an initial stage, all the combinations of items that have fractional transaction support above *minsup* are generated. All such combinations are denominated *large itemsets*. Next, given a large itemset S, generate rules of the type $S\text{-}X \Rightarrow Y$ for each $X \subset S$. Once these rules are generated, only those rules above a *minconf* may be retained.

A fast and popular algorithm for determining large itemsets called *Apriori* is presented in (Agrawal and Srikant, 1994). This algorithm includes a number of passes over the database and each one finds the set of frequent itemsets that satisfy the minimum support requirement.

4.2 Assumptions as association rules

Let $S=[E\|P]$ be the system and assume that we are model checking the component *P*. The process to automatically generate assumptions of the component *P* from the environment *E* is a problem that can be re-formulated on the basis of the discovering association rules, considering as the set of transactions the set of all possible configurations of *E* computed in the composition phase, *Global'*(*E*), and the set $(\Omega_E \cap \Psi_P) \cup \{mS\}$ as the items, where $\Omega_E \cap \Psi_P$ is the set of output interface events of the environment *E* that communicates with the component *P* and *mS* is the microstep counter. An assumption for *P* is an association rule between its input interface events and the instant (microstep), whichever occurs. The process to automatically derive assumptions can be stated as:

- **Large Eventsets**: Determine in *Global'*(*E*) the set of events (*eventsets*) with support greater than *minsup* (*large eventsets*). We can do this using the *Apriori* algorithm.
- **Assumption Generation**: Use the large eventsets to generate association rules with a confidence factor greater than *minconf*. These rules are the assumptions for the component *P*.

The assumptions, as we conceive them, must explain behaviours that are *ever-present* in the environment, so that the minimum confidence factor (*minconf*) must be *always 100*%. Similarly, the minimum support (*minsup*) in an assumption is not significant; even though an assumption is obtained only for one configuration in the component, the assumption is valid. Thus, erasing *minconf* and *minsup* enables full automation and simplifies the assumption generation process.

The above considerations have an important effect on the process of modular verification. If we obtain assumptions with respect to component *Q* from component *P*, it is not necessary to verify the obtained assumptions (assumption discharge) in *P* since these assumptions are generated as association rules with a confidence factor of 100%. That is, the obtained assumptions are guaranteed by their own generation method. We explain this process in detail in the next section.

4.3 Assumption generation

According to the method presented in the previous section, the first stage consists in computing the *large eventsets*, or sets of events with support above a threshold (*minsup*). However, we have shown that this factor is not significant in our process of assumption generation. Thus, all sets of events that occur in a configuration are large eventsets. A direct consequence of this consideration permits us to determine the large eventsets in just one pass over the set of configurations, thus avoiding the multiple explorations of the original algorithm (Agrawal and Srikant, 1994). Moreover, and more importantly, the counting of support in each eventset can be performed *on the fly* with the computation of the configurations. These ideas are summarized in the next algorithm.

```
Input: A configuration $C_j$ / $(C_i,C_j) \in \rightarrow^m$ for some $C_i \in$ Global'(E)
Output: Set $H_j$ of eventsets (large eventsets) in $C_j$
Procedure:
    $H_j = SubSets(C_j)$                      //eventsets in the configuration $C_j$
    $\forall h_k \in H_j$, $h_k$.support = $h_k$.support + 1   //update eventsets support
```

When a new configuration is created in *Global'*(*E*), the set of all possible subsets of events in this configuration C_j is computed (*SubSets* (C_j)). The *SubSets*(C_j) function can be defined as the power set of C_j excluding the empty set. At the same time, the corresponding support for each eventset is incremented. By applying the algorithm to each generated configuration in *Global'*(*E*), at the end of the process we have the set of eventsets (and its supports). To illustrate this approach, Table 1(a) displays the set of configurations of the component in Fig. 2, considering *OPEN* and *CLOSE* events as the output interface events. The set of eventsets corresponding to configuration 2 (with their respective cumulative support) is given in Table 1(b).

Table 1 (a) Configurations of the component in Fig.2

Configuration	OPEN	CLOSE	*mS*
1) OFF,CLOSED	0	0	0
2) ON,CLOSED	0	0	1
3) ON,OPENED	1	0	2
4) ON,OPENED	0	0	0
5) OFF,OPENED	0	0	1
6) OFF,CLOSED	0	1	2

(b) Eventsets for the configuration 2 (H_2)

EventSets	Support (Cum.)
OPEN=0	5
CLOSE=0	5
mS=1	2
OPEN=0 CLOSE=0	4
OPEN=0 *mS*=1	2
CLOSE=0 *mS* = 1	2
OPEN = 0 CLOSE = 0 *mS*=1	2

A similar observation can be made with respect to assumption generation. Since the confidence factor in an assumption is 100%, a necessary condition for two eventsets to form an assumption is that both have the same support. We thus avoid computation of the ratios between the support of eventsets and the test with *minconf* and the multiple phases to obtain rules with *1, 2...n* events in the consequent (Section 4.1). The algorithm that produces assumptions starting from the set of eventsets is shown next.

Input: Set of eventsets *H* of the environment *E*
Output: Assumptions from *E*
Procedure:

```
    $\forall h_i,h_j \in H,\ i \neq j,\ |h_i| \geq 2$
        if ( ($h_i$.support = $h_j$.support) $\wedge$ ($h_j \subset h_i$) )
                generate assumption $h_j \Rightarrow (h_i - h_j)$
```

Table 2 displays the generated assumptions related to the first microstep obtained by applying the above algorithm to the set of eventsets in Table 1(b).

Table 2 Generated Assumptions from the eventsets in Table 1(b)

Assumptions
mS=1 ⇒ OPEN=0
mS=1 ⇒ CLOSE=0
mS=1 ⇒ OPEN = 0 CLOSE=0

4.4 Macrostep normalization

In general, the macrosteps computed in a component do not have the same number of microsteps (*macrostep length*). This situation can affect the assumptions generated, introducing false behaviours in the environment. Suppose that the component environment executes two macrosteps with a length of two and three microsteps, respectively, and the goal is to observe the behaviour of an output interface event *a*. Suppose also that the set of computed configurations is:

Macrostep 1: $(mS=1,a=0) \rightarrow (mS=2,a=1)$

Macrostep 2: $(mS=1,a=1) \rightarrow (mS=2,a=0) \rightarrow (mS=3,a=1)$

Direct application of the assumption generation algorithm produces the assumption $mS=3 \Rightarrow a=1$, which indicates that in the third microstep, the environment produces the event *a* for all the macrosteps. However, it can easily be observed that this assumption is not real, because in the first macrostep, the event *a* does not take any value in the third microstep.

The way to solve this problem is to impose the condition that all the computed macrosteps have the same length. We call this process *macrostep normalization* and it consists in generating stuttering configurations to fill the affected macrostep. The events in the stuttering configurations have the value 0, but the microstep counter has the actual value corresponding to the microstep. In the above example, the first macrostep normalization consists in adding a stuttering configuration for the third microstep ($mS=3,a=0$). With this configuration in the first macrostep, the false assumption $mS=3\Rightarrow a=1$ is not obtained.

5 Application

In order to evaluate the approach and algorithms presented in this paper, we applied them to a non-trivial case study frequently used in the formal methods literature in general, and in that of formal verification in particular, namely the steam boiler specification problem (Abrial et al., 1996). We first present the set of tools used in our experimentation to then go on to describe the case study and the components into which the system has been broken down. The automatically generated assumptions are then described in detail. Finally, the analysis and results of modular verification of components with these assumptions are given and compared with the verification without assumptions and with manual assumptions.

5.1 Experimentation framework

The basic common integrated environment for specification and modular verification that we use is specified in (Tuya et al, 1997a). We complete this below, including the new support for the automatic assumptions explained in this paper (Fig. 9).

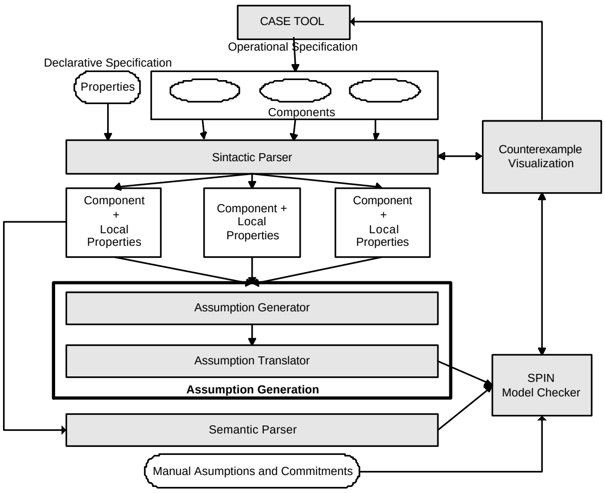

CASE TOOL
Operational Specification
Declarative Specification
Properties
Components
Sintactic Parser
Counterexample Visualization
Component + Local Properties
Component + Local Properties
Component + Local Properties
Assumption Generator
Assumption Translator
Assumption Generation
SPIN Model Checker
Semantic Parser
Manual Asumptions and Commitments

Fig. 9: Tools Integration for Assumption Generation

The specification of the system is divided into *operational* and *declarative* modes. The operational model is performed using a CASE tool. The *declarative* specification consists in the properties that the system must achieve. In order to perform modular verification, the system is broken down into separate parts and then compiled into synchronous reactive components (including the local properties) using a *syntactic parser*. The goal is to obtain an intermediate format independent of the CASE and model-checking tool (de la Riva et al, 2000). The components and their properties are then translated into the language of the model checker using a *semantic parser*. Assumptions are generated automatically from the components by means of the techniques and the algorithms detailed in this paper and translated into the language of the model-checking tool. The component is then verified with respect to each local property with the automatically generated and/or manually specified assumptions. To debug the specification (in the case of false in any property), the counterexample may be visualized in the original graphical language of the component specification. The verification tool we use is the SPIN model checker (Holzmann, 2003) version 3.4.8.

5.2 The steam boiler system

The *Steam Boiler Specification Problem* (Abrial et al, 1996) is a classical example used in the verification and specification literature with different notations and tools. The behaviour of the control program of the steam boiler is not trivial, as the requirements lay emphasis on a fault-tolerant behaviour of the system.

The system (Fig. 10) consists of the following elements: the steam boiler with a valve to evacuate water, a device to measure the amount of water in the steam boiler, four pumps to inject water into the boiler, their respective controllers, a device to measure the output steam from the boiler, an emergency stop switch and the message transmission system between the steam boiler and the control program.

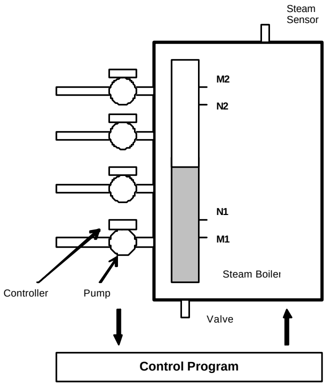


Fig. 10: The Steam Boiler

The normal operating cycle of the control program consists in reception of the messages sent by the physical units, analysis and processing of these messages and transmission of the messages to the physical units, its main task being to maintain the water level between two levels, *M1* and *M2*. Above *M2* and below *M1*, the system enters a fault state and the emergency stop must be activated by the desk operator. There are also two security levels, *N1* (*M1<N1*) and *N2* (*N2<M2*). When these are reached, the system must inject or evacuate water. Thus, the only way to control the water level in the steam is by opening or closing the pumps. The control program runs the following operational modes: *initialisation*, which represents the start-up of the control program; *normal*, which is the standard operating mode maintaining the water level between *N1* and *N2* when all devices work correctly; *degraded*, the control program tries to maintain a satisfactory level in the boiler when a failure exists in any physical unit (except for water measurement devices); *rescue*, similar to the degraded mode, though the failure is detected in the water measurement unit; and *emergency stop*, when the water levels reach *M1* or *M2*, or when a failure is detected in any vital unit.

5.3 Specification and verification

The high-level subsystem diagram is illustrated in Fig. 11. The external events (messages) sent and received by the control program are represented using uppercase labels. The remainder are internal events that communicate the subsystems.

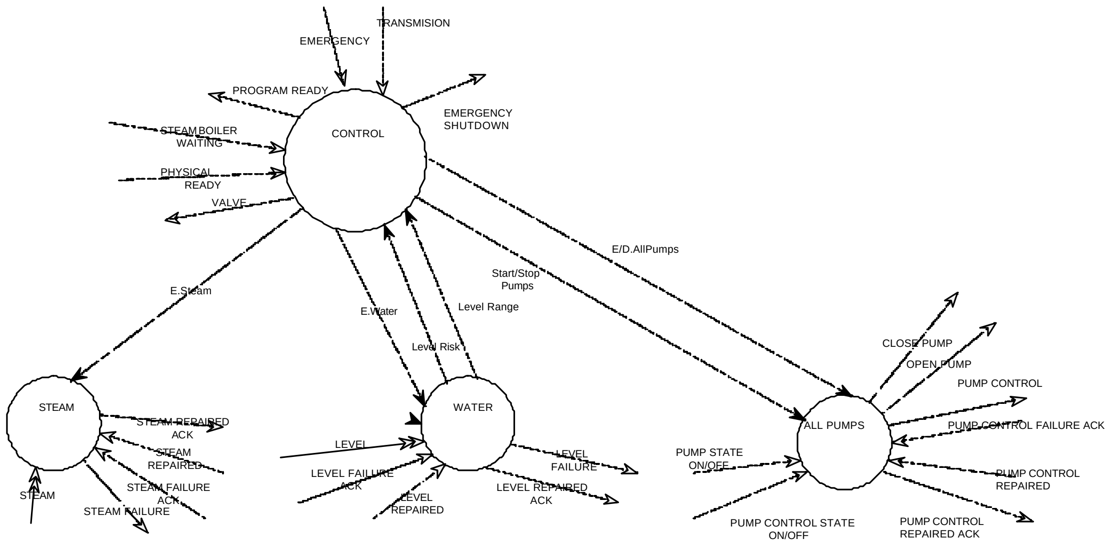


Fig.11: Subsystems of the Steam Boiler Control Program

The *CONTROL* subsystem maintains the different operational modes and launches the commands to initiate the opening or closing of pumps (events *StartPump* and *StopPump*, respectively) in accordance with the water level (event *LevelRange*) detected. It also enables and disables the remaining processes using the events labelled *E.process* and *D.process*, respectively. The *ALLPUMPS* subsystem controls the behaviour of the four pumps and their corresponding controllers and is responsible for sending opening and closing messages to each physical pump. The *WATER* subsystem determines the amount of the water in the boiler (*LevelRange*) and detects internal failures in the unit (*LevelRisk*). Finally, the *STEAM* subsystem controls the steam output from the boiler. In turn, the behaviour of each component is detailed using a set of primitive processes (specified as FSMs).

The declarative specification was obtained from the textual description of the system in (Abrial et al, 1996), where the requirements are stated in relation to the operational mode of the steam boiler. We translated these requirements into properties and specify them in propositional logic[1]. In total, the declarative specification consists of *55* properties.

As the four pumps are identical instances, we planned to perform the verification of all properties over the system in an incremental fashion. Firstly, we considered the system formed by one single pump to then go on to incrementally add a pump to the specification. However, verification was not possible for any property even considering the system composed of one pump (the system is composed of 26 FSMs). In fact, either SPIN produced an "*out of memory*" situation or was stopped when the verification time was greater than five days.

5.4 Component verification and assumption generation

To avoid the state explosion, we will divide the steam boiler specification into components and will perform local verifications over these using the assume-guarantee paradigm. Modular verification will be conducted by partitioning the system into four components, each one corresponding to a subsystem in the specification in Fig. 11. In turn, the global properties will be modularised in local properties, enabling each one to be verified over the proposed components. For example, consider the global property φ "*in*

[1] The translation process from the component to the SPIN language (PROMELA) states that the properties are checked at the end of each macrostep. For this reason, even when SPIN supports LTL properties, it is not necessary to specify them in temporal logic. They will be encoded in SPIN with the PROMELA sentence `assert` (Tuya et al, 1997b).

the Normal mode, if the water level is not within the permitted limits, the control program must start an emergency stop mode" encoded as:

$$((ControlRunMode == Normal)\ \&\&\ (LEVEL{=}{=}LessM1\ ||\ LEVEL{=}{=}GreaterM2)) \rightarrow ControlModes{=}{=}EmergencyStop$$

This property references elements in the component *CONTROL* (the processes *ControlRunMode* and *ControlModes*) and the component *WATER* (the event *LEVEL*). This property can be broken down into the following:

φ_1: $LevelRisk \rightarrow (LEVEL == LessM1\ ||\ LEVEL == GreaterM2)$

$\varphi_{2:}$ $(ControlRunMode == Normal\ \&\&\ LevelRisk) \rightarrow ControlModes == EmergencyStop$

It is clear that $\varphi_1 \wedge \varphi_2 \rightarrow \varphi$ and both properties can be analysed separately in each component.

All properties of the original specification were modularised following the above procedure. Table 3 shows the components along with the number of processes and properties for modular verification.

Table 3: Components for modular verification

Component	Number of FSMs	Number of Properties
ALLPUMPS (one pump)	11	20
CONTROL	5	19
WATER	5	10
STEAM	5	6

We will first check each component assuming a "universal environment" (the environment model is not provided), then with manual assumptions based on the engineer's knowledge and finally with automatically generated assumptions using the algorithms described in this paper.

5.4.1 CONTROL component

This component maintains the operational modes of the control program on the basis of the events received from the component *WATER* (*LevelRange* and *LevelRisk*) while sending the corresponding events (*StartPump* and *StopPump*) to the component *ALLPUMPS*. In the initialisation mode, *CONTROL* enables the remainder of the components.

We performed the verification of this component assuming a universal environment, i.e. the input interface events (*LevelRange* and *LevelRisk)* can take any value in any microstep. Four properties were found to be false. Two of these relate to the initialisation mode. The first one states that when the water level is high (*LevelRangeHigh*), the valve must be opened (*VALVE=1*) and the second, that when the level is low (*LevelRangeLow*), the component must send the stop event to the pumps (*StopPumps*). Analysis of the counterexamples returned by the model checker shows that the input interface events that indicate the water level could be received at the same time, and thus the component may react by opening the pumps, when in fact it should be opening the evacuation valve. Obviously, this environment's behaviour is unrealistic and the error can be removed by performing the environment assumption that the water level events cannot be received at the same instant. After verifying the component once more with this manual assumption, the two properties are true and, after discharging the assumption, it is true over the environment (*WATER* component).

A similar situation is produced in the third false property relative to the interface input events *LevelRisk* and *LevelRange*. The assumption that indicates that both events do not have the same value at the same instant resolves the problem and is true in *WATER*.

Finally, the last false property indicates that when the control mode is normal and a failure water level event (*LevelRisk*) has been received in the preceding instant, the control program must immediately evolve to an *EmergencyStop* state. Yet again, a counterexample analysis can help us to determine the cause of the error. However, its interpretation is more difficult than in the previous cases and after several attempts and simulations, we were not able to fix the error and manually specify a correct assumption.

To solve this situation and help us to decide whether the error is a false positive or whether it is effectively a specification mistake, we automatically generate assumptions (Table 4) to observe the behaviour of the input interface events (*LevelRisk* and *LevelRange*).

Table 4: Automatically generated assumptions for the CONTROL component from the WATER component

1	LevelRangeLow ⇒ LevelRangeHigh=0 && LevelRangeMiddle=0 && LevelRisk=0
2	LevelRangeHigh ⇒ LevelRangeLow=0 && LevelRangeMiddle=0 && LevelRisk=0
3	LevelRangeMiddle ⇒ LevelRangeLow=0 && LevelRangeHigh=0 && LevelRisk=0
4	LevelRangeRisk ⇒ LevelRangeLow=0 && LevelRangeHigh=0 && LevelMiddle=0
5	*mS*≤2 ⇒ LevelRangeHigh=0 && LevelRangeMiddle=0 && LevelRangeLow=0 && LevelRisk=0

The four first assumptions are the same as those manually specified to solve the false positives. They indicate that input interface events are not sent at the same microstep. The last assumption states the behaviour of the environment in concrete microsteps: input interface events are not sent until the third microstep. After executing the model checker once more with these new assumptions, all the properties in the component *CONTROL* are true. Specifically, the fourth property, which returned false under the universal environment, is now true because of the introduction of the fifth assumption. We can thus conclude that the initial error was a false positive not caused by an incorrect system specification but by an incorrect specification of the component environment. Note that this assumption is very difficult to obtain manually, as it involves semantic concepts not easily accessible to the user.

5.4.2 Other components

The component *ALLPUMPS* maintains the state of the four pumps and their respective controllers, sending open and close orders to physical devices. It is related to the component *CONTROL* by means of the events *StartPumps* and *StopPumps* (Fig. 11). As the component environment is the same whether considering one or more pumps, for the moment we will take the component formed by one pump. Furthermore, we will provide additional considerations on verification with more pumps.

As in the previous component, *ALLPUMPS* was first verified without assumptions and then with manually specified assumptions related to incompatible reception of the input interface events (*StartPumps⇒StopPumps=0* and *StopPumps⇒StartPumps=0*). In both cases, one property was found to be false: *"if the pump or its controller is in a failure state, the component does not send opening or closing orders to physical devices"*. The counterexample shows that the pump is in a failure state and a close pump event is sent to the physical device. Initially, this result was surprising, because all the transitions in the process responsible for enabling the respective open and close orders are safeguarded by the condition *PumpFailureStatus!=Failure*.

We automatically generate the assumptions for the component *ALLPUMPS* from the component *CONTROL* (Table 5).

Table 5: Assumptions for ALLPUMPS component from the CONTROL component

1	StartPumps || E.AllPumps ⇒ StopPumps=0
2	StopPumps || E.AllPumps ⇒ StartPumps=0
3	E.AllPumps ⇒ D.AllPumps=0
4	D.AllPumps || StartPumps || StopPumps ⇒ E.AllPumps=0
5	*mS*==1 ⇒ StopPumps=0 && StartPumps=0
6	*mS*>1⇒ E.AllPumps=0
7	*mS*>5 ⇒ StartPump=0 && StopPumps=0 && D.AllPumps=0

Assumptions 1 to 4 show the incompatible reception at the same microstep in the component *ALLPUMPS* of the input interface events. The remaining assumptions are related to specific microsteps and indicate that in the first microstep the start/stop events cannot occur (Assumption 5), the enable event *E.AllPumps* is never sent after the first microstep (Assumption 6) and the start/stop events and the disable event are never sent after the fifth microstep (Assumption 7). The component *ALLPUMPS* was checked again using this set of generated assumptions and all the properties returned true.

The WATER component determines the amount of water in the boiler and the STEAM component controls the amount of output steam. In both components, all the properties were true without assumptions and the automatic assumptions computed did not improve the result of the verification process.

5.5 Performance

In this section we will show the results of the verification of each component of the system. For each one, we perform verification considering the universal environment (no assumptions), manual assumptions and automatically generated assumptions. All experiments were performed on a Compaq Proliant with 933 MHz. and 1 Gbyte of main memory using the SPIN model checker version 3.4.8.

5.5.1 CONTROL component

Table 6 displays the results of the SPIN performance in terms of time (column *t*, in seconds) and space (column *mem*, in Mbytes) during the verification of the local properties in the component *CONTROL* with the universal environment (no assumptions), manual assumptions and the automatically generated assumptions specified in Table 4. Shaded performance data represents a false (positive) property. The last rows present a summary of the total verification time (the improvement factor with respect to the universal and manual assumptions, respectively, is given in brackets in the case of automatic assumptions), the number of assumptions used, the number of false positives obtained and the CPU time of assumption discharge and assumption generation.

Table 6: Results of the verification of the CONTROL component

	Universal Environment		***Manual Assumptions***		***Automatic Assumptions*** (Table 4)	
Property	***t***	***mem***	***t***	***mem***	***t***	***mem***
1	135	3.6	30	2,3	9	1.9
2	135	3.6	30	2.3	9	1.9
3	21	2.0	19	2.0	13	2.2
4	135	3.6	30	2.3	9	1.9
5	135	3.6	30	2.3	9	1.9
6	196	3.9	41	2.5	12	2
7	154	3.6	34	2.3	10	1.9
8	135	3.6	30	2.3	9	1.9
9	135	3.6	30	2.3	9	1.9
10	154	3.6	34	2.3	9	1.9
11	154	3.6	34	2.3	10	1.9
12	139	3.6	30	2.3	9	1.9
13	178	3.8	40	2.5	12	2.1
14	150	3.6	35	2.3	9	1.9
15	139	3.6	30	2.3	9	1.9
16	139	3.6	30	2.3	9	1.9
17	23	2.1	51	3	16	2.4
18	14	2.0	82	3.8	27	2.9
19	16	2.0	12	2.0	34	3.5
Total Verification (sec.)	**2 017**		**592**		**229 (8.4) (2.4)**	
Number of Assumptions	**0**		**4**		**5**	
Number of False Positives	**4**		**2**		**0**	
Assumption Discharge (sec.)			**664**			
Assumption Generation (sec.)					**804**	

In general, the CPU time and memory use of the model checker decreases with the incorporation of new assumptions in the verification of each property. This result is to be expected a priori. It is actually due to the fact that the environment is limited by the assumptions and the model checker will handle smaller state spaces. For example, in the verification with the universal environment, the model checker must explore all possible combinations of the interface events *LevelRange* and *LevelRisk* in all the microsteps and in any order. The introduction of assumptions in these events will limit all the possible values, the order of these and even the microsteps in which they can occur.

The automatically generated assumptions substantially improved the performance of the model checker for up to a factor of 8.4 with respect to the universal environment and of 2.4 with respect to manually specified assumptions. In the latter case, this is due to a greater number of assumptions and the fact that they represent the real behaviour of the environment better. Moreover, it is only possible to verify all the properties in the component without producing false positives using the automatically generated environment. This result will be more important if the number of properties to verify is incremented or if we perform a regression verification over previously verified component. This is because if a new property is added, it is not necessary to compute the environment again (except when the environment is also changed).

It should also be noted that the “manual time” of assumption specification is not included (in the case of the manual approach). Obviously, this effort is difficult to measure and the discovery of an adequate assumption by the engineer is a complicated task that requires significant mental effort. From this point of view, the automatic approach is a useful tool that reduces computational resources and user guidance.

5.5.2 Other components

We now perform the same experiments for the component *ALLPUMPS* (with one pump) summarized in Table 7.

Table 7: Summarized results of the *ALLPUMPS* component with one pump

	Universal Environment	*Manual Assumptions*	*Automatic Assumptions* (Table 5)
Total Verification (sec.)	157	60	13 (12.1) (4.6)
Number of Verified Prop.	19	19	19
Number of Assumptions	0	2	7
Number of False Positives	1	1	0
Assumption Discharge (sec.)		828	
Assumption Generation (sec.)			140

As in the component *CONTROL*, the model checking time of all properties is lower with the automatic environments (an improvement factor of 12.1 and 4.6 with respect to the universal environment and manual assumptions, respectively) and all the properties were verified without false positives.

In order to check how our approach affects the performance of the model checker in the verification of systems that are greater in size and complexity, we scaled up the component *ALLPUMPS* by incrementing the number of pumps in the specification. The summarized results with two pumps are given in Table 8.

Table 8: Summarized results of the *ALLPUMPS* component with two pumps

	Universal Environment	*Manual Assumptions*	*Automatic Assumptions* (Table 5)
Total Verification (sec.)	246 044	213 087	19 891 (12.3) (10.7)
Number of Verified Prop.	18	18	19
Number of Assumptions	0	2	7
Number of False Positives	1	1	0
Assumption Discharge (sec.)		828	
Assumption Generation (sec.)			140

To verify the component without assumptions and with manual assumptions, we need a machine time of approximately 78 and 59 h, respectively. With the automatic approach, this time is reduced to 5 hours approximately. That is, the speed-up obtained was a factor of 12.3 and 10.7, respectively.

Another important consideration is related to the number of properties verified. With the automatically generated assumptions, all the properties can be verified. However, with universal and manual approaches, one property returns an *out of memory* SPIN message. This property was verified with the automatic approach in approximately 2 h.

The component with three and four pumps cannot be verified: for all properties, the model checker returns an *out of memory* response (a different partition into components would be performed in order to verify each component). This result is expected if we consider the results obtained from one (Table 7) and two pumps (Table 8) due to the combinatorial state explosion.

In the case of the WATER and STEAM components, the resource consumptions of the model checker were small (<1 s, <1.5 Mbyte) and similar with respect to the universal environment, manual and automatic assumptions. This is due to both these components having a reduced input interface (only enable and disable events) and consequently the obtained assumptions do not contribute to restricting the component environment.

6 Discussion

The approach presented in this paper enables assumptions to be obtained automatically over a component making feasible assume-guarantee modular proofs. Owing to the fact that we perform an exhaustive exploration of all the microsteps in the environment computation (including those derived from non-determinism), the obtained assumptions explain environment behaviours of the input interface events, some of which are difficult for the engineer to guess, thus avoiding the need for their manual specification. The following results of this approach may be noted:

- The main advantage of our technique is that it removes the time-consuming and error-prone task of the manual specification of assumptions that is always necessary in modular verification.
- Consequently, it is not necessary to perform assumption discharge onto the environment, thus reducing the effort needed in the verification process.
- An important feature is that it permits the discovery of false positives, or properties that, though in fact true, are returned as false by the model checker due to an incorrect or unrealistic specification of the environment. Because the technique does not generate all possible assumptions, we cannot ensure that it detects all possible false positives, and hence it is not totally complete. However, the technique is sound because the assumptions do not insert additional behaviours and hence ensure that every component errors appear.
- As a result, the state space that the model checker must deal with is smaller and hence its performance is improved. In all the cases analysed, the performance of the model checker using automatic assumptions is superior in a number of factors with respect to the assumptions specified by the engineer or when a universal environment is considered (no assumptions). This is because there are more assumptions and they represent the behaviour of the environment more exactly, limiting the arrival of interface events in certain microsteps, avoiding the reception of inconsistent combination of events and reducing non-determinism in the component to verify.

The problem of assumption generation is also addressed in a recent approach to automated assume-guarantee procedures (Giannakopoulou et al., 2005). However, the method employed is not independent of the component and the property to verify. Our approach computes assumptions independently from the component and property under analysis. That is, if the component or property change or if we add new properties to verify, it is not necessary to calculate the assumptions again. From this point of view, our method is especially adequate in *incremental* and *regression* verification practices. In fact, it is normal during component verification for any property to be false because of an incorrect component specification. In this case, the component is adequately modified and all properties are once more verified (regression verification). It is not necessary to compute the assumptions again for each property verification because they are the same as the ones obtained in the first verification. A similar observation can be made for the case when new properties are added or completed in an interactive and incremental fashion. If the environment has been previously computed, no new assumptions are needed for each new property.

Though the method is applied to the SPIN model checker, it is can be easily adapted to other finite state verification tools. As the assumption generation is made independently of the verification tool, the only changes are those related to the component and assumption coding in the target tool

Although, we focus on state transition systems, assume-guarantee reasoning may be applied to other formalisms such as for example, process algebras (Rahanabu and Rehof, 2001). In the context of our approach, the only imposed restriction is that the underlying finite state model is specified in terms of reactive semantics.

The experiments and experience gained with automatically generated assumptions and their application to assume-guarantee reasoning opens up interesting lines of research and improvements to our technique. We cite a few below.

- The main bottleneck of the approach is to be found in the composition phase, due to a possible configuration explosion and computational resources consumptions. Although we proposed a heuristic to reduce the state space, the adoption of new methods of state space coding, for example using BDD (Bryant, 1986), and/or state space exploration, for example using partial order reductions (Peled, 1994), would be useful.
- The method was applied to the verification of safety properties. In general, our approach is also appropriate for liveness properties due to the independence of the component and property. However, greater expressive power is needed in the assumptions in order to apply the method to the verification of certain types of liveness properties such as livelocks. So as to be able to detect this undesired behaviour, we need to generate assumptions in such a way that an interface event appears only once, at most, in the same microstep. Within the same context, an interesting line of research would be to search for new methods for coding the assumptions; for example with temporal logic where the expressive power is greater than in association rules.
- Automatic reduction of the generated assumptions: in general, some assumptions produced could be redundant, in the sense that certain assumptions are included or can be derived from others. In the extraction step, these could be detected using similar methods to those used in association rules techniques (Aggarwal and Yu, 1998).

7 Related work

The modular or compositional approach is a natural and effective technique for solving the problem of verification of large complex systems. In recent decades, various authors have proposed theoretically well-founded and sound frameworks for the modular reasoning of reactive systems.

Traditionally, much of the work on compositional or modular analysis and verification has been carried out in *compositional minimization,* i.e. to obtain a reduced version of the environment in each component that exactly characterized the visible behaviours of the component under analysis (*interface process*) so as to then perform the composition between the component and the interface process. As the interface process is less complex than the concrete (removed) component, verification may be feasible by minimizing the state space explosion.

In the work of Clarke et al. (Clarke et al., 1989), the interface process is constructed removing those transitions in the component that are not present in the alphabet of the environment component. Graff et al. (1996) develop a method that includes said technique, and in which the interface process is specified by the user. Both methods produce large state spaces if most of the states of the systems are not equivalent. This approach is further developed and partially automated in (Cheung and Krammer, 1996) by means of an algorithm to automatically construct the interface process. However, the automatically generated environments must be completed by the user with stubs because they do not sufficiently limit the behaviour of the environment. Grumberg and Long (1994) describe a framework for compositional verification for temporal model checking based on assume-guarantee reasoning in which environmental assumptions are explicitly and manually provided to the component. Since we do not have sufficient space to describe all the other pertinent studies equitably, we refer the reader to the book by de Roever and colleagues (de Roever et at., 2001), which provides an excellent introduction to the state of the art in the compositional methods for verification.

The practical use and application of these techniques has not, however, evolved at the same speed as the underlying theories. As Shankar observes (Shankar, 1997), the inference systems for compositional verification have been studied more than applied. From our point of view, this is due to the lack of an adequate automated support for this style of reasoning.

Recent developments in OO-design and in component-based software engineering have provided new advances in automated formal techniques for specification and verification. In this respect, the studies by de Alfaro and Henzinger (2001a,b); Alur et al. (1999) contribute to automating the modular verification. Some of these theories and techniques have been implemented in the MOCHA toolkit (Alur et al, 1998), which provides automated support for modular reasoning with specifications in temporal logic.

When using assume-guarantee modular reasoning, a way is needed to specify the more adequate environment assumptions (Henzinger et al, 1998) that sufficiently limits the environment while not producing false violations (false positives). For this reason, research in techniques and methods to automatically generate the component environment is very helpful. However, although the problem is not novel (it is in fact as old as assume-guarantee reasoning), the work in automated assumption generation is very limited and very recent.

An approach to assumption generation is presented in (Inverardi et al, 2000), though it is restricted to checking the compatibility between components and modular checking of deadlock freedom properties. In the field of shared-memory communication models, Henzinger et al (2003) have presented a framework for thread-modular abstraction refinement in which thread assumptions are refined in an iterative fashion and applied to the detection of race conditions. The work of Flanagan and Qadeer (2003) also focuses on programs that communicate through shared variables, but does not use abstractions.

The closest to our work is the approach described in (Giannnakopoulou et al, 2005), in which an algorithm to automatically generate the *weakest* assumption is presented. The method restricts the environment no more and no less than is necessary for a component to satisfy a given property. More specifically, the proposed technique returns one of following results: the component satisfies the property for all environments, the component violates the property for any environment or an assumption is returned that precisely characterizes those environments in which the component satisfies the property. A framework for performing assume-guarantee in an incremental and fully automatic fashion is presented in (Cobleigh et al, 2003). The assumptions are computed using a learning algorithm. Initially, the obtained assumptions are approximated, but become progressively more precise by means of the analysis information in the obtained counterexamples. The main difference with respect to the previous approach is that if the computation runs out of memory (due to the state space of the component being too large), the assumptions computed at that moment are valid. Automated support for this incremental framework is described in (Tkachuk et al. 2004). The work of Barriger et al (2003) extends the previous approach employing symmetric assume guarantee rules. Chaki et al. (2005) have applied a similar learning paradigm to automate assumption-guarantee reasoning for simulation conformance between finite systems and their specification. The relation between system decomposition and modular verification is addressed in (Cobleigh et al. 2004), where the authors examine the advantages of assumption-based verification over monolithic verification in all two-way decompositions of the system using this assumption generation technique.

Unlike our approach, the above methods are dependent on the property and the component to verify. For this reason, the algorithms must be applied to each property to verify, and any change in the component (or property) needs a new verification. However, the approaches are very interesting when the environment is not given or is unknown.

8 Conclusions

In this paper we present a method for model-checking components based on assume-guarantee reasoning with automatically generated assumptions. Our approach generates a set of environmental rules that explain the behaviour of interface events and which can be used as assumptions of the components, thus avoiding their manual specification and consequently reducing the user guidance needed in modular verification. The technique is especially relevant in detecting false positives, due to there being more obtained assumptions.

This ability to generate assumptions also significantly affects the performance of the model checker. The assumptions reflect the behaviour of environmental events more precisely and so the number of states related to these events that the model checker must explore is also limited.

Due to the way in which assumptions are generated on the basis of the environment behaviour, if the component or property changes, the assumptions do not need to be computed again. This aspect is important in regression verifications, as the environment must be computed only once.

Application to a non-trivial system shows the viability of the approach in detecting false positives and in reducing the consumption of resources of the model checker in particular, and of the complete verification process in general. This experience opens up possible improvements to the approach, particularly as regards the expressivity of the generated assumptions and those aimed at reducing blow-up in the computation of environment configurations.

Acknowledgements

This work was funded by the Ministry of Science and Education (SPAIN) under the projects IN2TEST (TIN2004-06689-C03-02) and ARGO (TIC2001-1143-C03-03).

References

Abadi, M., Lamport L., 1995. Conjoining Specifications, ACM Transactions on Programming Languages and Systems 17 (3), 507-534.

Abrial, J.R., Borger, E., Langmaack, H., 1996. Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, Volume 1165 of LNCS. Springer-Verlag.

Aggarwal, C.C., Yu, P.S., 1998. Online Generation of Association Rules. In: Proceedings of the International Conference on Data Engineering, pp. 402-411.

Agrawal, R., Imielinski, T., Swami, A., 1993. Mining Association Rules between Sets of Items in Large Databases. In: Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 207-216.

Agrawal, R., Srikant, R., 1994. Fast Algorithms for Mining Association Rules. In: Proceedings of the International Conference on Very Large Databases, pp. 487-499.

Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S., Tasiran, S., 1998. Mocha: Modularity in Model Checking. In: Proceedings of 10th International Conference on Computer Aided Verification, Volume 1427 of LNCS, pp 521-525, Springer-Verlag.

Alur, R., de Afaro, L., Henzinger, T.A., Mang, F.Y.C., 1999. Automating Modular Verification. In: Proceedings of the 10th International Conference on Concurrency Theory, Volume 1664 of LNCS, pp. 82-97. Springer-Verlag.

Barringer, H., Giannakopoulou, D., Pasareanu, C.S. 2003. Proof Rules for Automated Compositional Verification Through Learning. In Proceedings. of the Second Workshop on Specification and Verification of Component-Based Systems, pp 14–21.

Bryant, R.E., 1986. Graph-based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers, C-35 (8), 388-397.

Chaki, S., Clarke, E., Groce, A., Ouaknine, J., Strichman, O., Yorav, K. 2004. Efficient Verification of Sequential and Concurrent C Programs, Formal Methods in System Design, 25 (2-3), 129-166.

Chaki, S., Clarke, E.,Sinha, N., Thati, P. 2005. Automated Assume-Guarantee Reasoning for Simulation Conformance. In Proceedings of the 17th International Conference on Computer-Aided Verification, Volume 3576 of LNCS, pp. 534-547. Springer-Verlag.

Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J., 1998. Model Checking Large Software Specifications, IEEE Transactions on Software Engineering 24 (7), 498-520.

Chan, W., Anderson, R.J., Beame, P., Jones, D.H., Notkin, D., 2001. Optimizing Symbolic Model Checking for Statecharts, IEEE Transactions on Software Engineering 27 (2), 170-190.

Cheung, S.C., Kramer, J., 1996. Context Constraints for Compositional Reachability Analysis, ACM Transactions on Software Engineering and Methodology, 27 (2), 170-190.

Clarke, E.M., Long. D.E., McMillan, K.L., 1989 .Compositional Model Checking. In: Proceedings of the 4th Symposium on Logic in Computer Science, pp. 353-362.

Clarke, E.M., Grumberg, O., Peled, D.A., 1999. Model Checking, The MIT Press, Cambridge. MA.

Cobleigh, J., Giannnakipoulou, D., Pasareanu, CS., 2003. Learning Assumptions for Compositional Verification. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Volume 2619 of LNCS, pp. 331-346. Springer-Verlag.

Cobleigh, J., Avrunin, G., Clarke, L., 2004. Breaking Up is Hard to Do: An Investigation of Decomposition for Assume-Guarantee Reasoning. Technical Report UM-CS-2004-023, University of Massachusetts, Department of Computer Science.

de Alfaro, L., Henzinger, T.A., 2001a. Interface Automata. In: Proceedings of 8th European Software Engineering Conference, pp. 109-120.

de Alfaro, L., Henzinger, T.A., 2001b. Interface Theories for Component-based Design. In: Proceedings of EMSOFT 200: Embedded Software, Volume 2211 of LNCS, pp. 148-165. Springer-Verlag.

de la Riva, C. Tuya, J., de Diego, J.R., 2000. Translating SA/RT Models to Synchronous Reactive Systems: An Approximation to Modular Verification Using the SMV Model Checker. In: 3th International Conference of Perspectives of System Informatics, Volume 1755 of LNCS, pp. 493-502. Springer Verlag.

de Roever, W.P., de Boer, F., Hanneman, U., Hooman, J., Lakhnech, M.P., Zwiers, J., 2001. Concurrency Verification: Introduction to Compositional and Non-compositional Methods. Cambridge University Press.

Flanagan, C., Qadeer, S., 2003. Thread-modular Model Checking. In: Proceedings of the 10th SPIN Workshop, Volume 2648 of LNCS, pp 213-224, Springer-Verlag.

Giannakopoulou, D., Pasareanu, C., Barringer, H., 2005. Component Verification with Automatically Generated Assumptions, Automated Software Engineering, 12(3), 297-320

Graf, S., Steffen, B., Lüttgen, G., 1996. Compositional Minimization of Finite State Systems Using Interface Specifications. Formal Aspects of Computing 8 (5), 606-616.

Grumberg, O., Long, D.E., 1994. Model Checking and Modular Verification. ACM Transactions on Programming Languages and Systems 16 (3), 843-871.

Harel, D., Pnueli, A., Pruzan-Schmidt, J., Sherman, R., 1987. On the Formal Semantics of Statecharts. In: Proceedings of Symposium on Logic in Computer Science, pp. 54-64.

Harel, D., Naamad, A., 1996. The STATEMATE Semantics of Statecharts, ACM Transactions on Software Engineering and Methodology, 5 (4), 293-333.

Henzinger, T.A., Qadeer, S., Rajamani, S.K., 1998. You Assume. We guarantee: Methodology and Case Studies. In: Proceedings of the 10th International. Conference on Computer Aided Verification, pp. 250-265.

Henzinger, T.A, Jhala R, Majumdar R, Qadeer S., 2003. Thread-Modular Abstraction Refinement. In: Proceedings of the 15th International Conference on Computer Aided Verification. Volume 2725 of LNCS, pp. 262-274. Springer-Verlag.

Holzmann, G., 2003. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley.

Inverardi, P., Wolf, A.L., Yankelevich. D., 2000. Static Checking of System Behaviours Using Derived Component Assumptions, ACM Transactions on Software Engineering and Methodology, 9 (3), 239-272.

Jagadeesan, L.J., Puchol, C., Olnhausen, J.E.V., 1996. A Formal Approach to Reactive Systems Software: A Telecommunications Application in ESTEREL, Formal Methods in System Design, 8,(2), 123-151.

Jones, C., 1983. Tentative Step Towards a Development Method for Interfering Programs. ACM Transactions on Programming Languages and Systems 5(4), 596-619.

Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D., 1994. Requirements Specification for Process Control Systems, IEEE Transactions on Software Engineering, 20 (9), 684-707.

McMillan, K.L., 1993. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academics.

McMillan, K.L., 2000. Compositional Methods and Systems. Verification of Digital and Hybrid Systems, Volume F-170 of NATO Advanced Summer Institutes, pp. 138-151, Springer-Verlag.

Peled, D.A., 1994. Combining Partial Order Reductions with On-the-Fly Model Checking. In: Proceedings of the 6th International Conference on Computer Aided Verification, Volume 818 of LNCS, pp. 377-390, Springer-Verlag.

Pnueli, A., 1984. In Transition from Global to Modular Temporal Reasoning about Programs. In: Logic and Models of Concurrent Systems, Volume 13 of LNCS, pp. 123-144. Springer-Verlag.

Rahanabu, S.K., Rehof, J. 2001. A Behavioral Module System for the Pi-Calculus. In Proceedings of the Statatic Analysis Symposium, Volume 2126 of LNCS, pp. 375-394, Springer-Verlag.

Shankar, N., 1997. Lazy Compositional Verification. In: Compositionality: The Significant Difference – An International Symposium, Volume 1536 of LNCS, pp. 541-564, Springer-Verlag.

Sreemani, T., Atlee, J.M., 1996. Feasibility of Model Checking Software Requirements: A case study. In: Proceedings of the 11th Annual Conference on Computer Assurance, pp. 77-88, IEEE.

Tkachuk, O., Dwyer, M., Pasareanu, C.S., 2004. Automated Environment Generation for Software Model Checking. In: Proceedings of 18th IEEE International Conference of Automated Software Engineering, pp. 116-129.

Tuya, J., de la Riva C., de Diego, J.R. 1997a. CASE Support for Modular Verification of Synchronous Reactive Systems. In: *Proceedings of 2nd International Workshop on Formal Methods for Industrial Critical Systems*. pp. 125-137.

Tuya, J., de la Riva, C., de Diego, J.R., Corrales, J.A. 1997b. Dynamic Analysis of SA/RT Models using SPIN and Modular Verification. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 3, pp. 165-183.

Claudio de la Riva is an Assistant Professor of the Department of Computer Science, University of Oviedo, Spain. He has received the Ph.D. degree from the Department of Computer Science, University of Oviedo, Spain, in 2004. His research interests include Software Verification and Validation and Software Testing.

Javier Tuya is an Associate Professor of the Department of Computer Science, University of Oviedo, Spain. He has received the Ph.D. degree from the Department of Electrical Engineering, University of Oviedo, Spain, in 1995. He is member of the ACM, IEEE and the Computer Society. His research interests include Software Quality Assurance, Process Improvement, Verification and Validation and Software Testing.