Chapter 2
Efficient Refinement Strategy Exploiting Component Properties in a CEGAR Process

Syed Hussein S. Alwi, Cécile Braunstein, and Emmanuelle Encrenaz

Abstract Embedded systems are usually composed of several components and in practice, these components generally have been independently verified to ensure that they respect their specifications before being integrated into a larger system. Therefore, we would like to exploit the specification (i.e. verified CTL properties) of the components in the objective of verifying a global property of the system. A complete concrete system may not be directly verifiable due to the state explosion problem, thus abstraction and eventually refinement process are required. In this paper, we propose a technique to select properties in order to generate a good abstraction and reduce refinement iterations. We have conducted several preliminary experimentations which show that our approach is promising in comparison to other abstraction-refinement techniques implemented in VIS [1].

2.1 Introduction

The embedded systems correspond to the integration into the same electronic circuit, a huge number of complex functionalities performed by several heterogeneous components. Current SoC (System on Chips) contain multiple processors executing numerous cooperating tasks, specialized co-processors (for particular data treatment or communication purposes), Radio-Frequency components, etc. These systems are usually submitted to safety and robustness requirements. Depending on their application domains, their failure may induce serious damages and catastrophic consequences.

Therefore, it is important to ensure, during their design phase, their correctness with respect to their specifications. Errors found late in the design of these systems

S.H.S. Alwi (✉) • C. Braunstein • E. Encrenaz
Université Pierre et Marie Curie Paris 6, LIP6-SOC (CNRS UMR 7606),
4, place Jussieu, 75005 Paris, France
e-mail: syed-hussein.alwi@lip6.fr; cecile.braunstein@lip6.fr; emmanuelle.encrenaz@lip6.fr

J. Haase (ed.), *Models, Methods, and Tools for Complex Chip Design*, Lecture Notes in Electrical Engineering 265, DOI 10.1007/978-3-319-01418-0_2,

is a major problem for electronic circuit designers and programmers as it may delay getting a new product to the market or cause failure of some critical devices that are already in use. System verification using formal methods such as model checking guarantees a high level of quality in terms of safety and reliability while reducing financial risk.

The main challenge in model checking is dealing with the state space combinatorial explosion phenomenon. A strategy to overcome the state explosion problem is by performing abstraction. A method for the construction of an abstract state graph of an arbitrary system automatically was first proposed by Graf and Saidi [2] using Pvs theorem prover. Here, the abstract states are generated from the valuations of a set of predicates on the concrete variables. The construction approach is automatic and incremental.

In 2000, an interesting abstraction-refinement methodology called counterexample guided abstraction refinement (CEGAR) was proposed by Clarke and al. [3]. The abstraction was done by generating an abstract model of the system by considering only the variables that possibly have a role in verifying a particular property. In this technique, the counterexample provided by the model-checker in case of failure is used to refine the system.

Several tools using counterexample-guided abstraction refinement technique, like those implemented in the VIS model-checker, have been developed such as SLAM, a software model-checker by Microsoft Research [4], BLAST (Berkeley Lazy Abstraction Software Verification Tool), a software model-checker for C programs [5] and VCEGAR (Verilog Counterexample Guided Abstraction Refinement), a hardware model-checker which performs verification at the RTL (Register Transfer Language) level [6]. However, relying on counterexamples generated by the model checker as the only source for refinement may not be conclusive.

Recently, a CEGAR based technique that combines precise and approximated methods within one abstraction-refinement loop was proposed for software verification [7]. This technique uses predicate abstraction and provides a strategy that interleaves approximated abstraction which is fast to compute and precise abstraction which is slow. The result shows a good compromise between the number of refinement iterations and verification time.

An alternative method to get over the state explosion problem is the compositional strategy. The strategy is based on the assume-guarantee reasoning where assumptions are made on other components of the systems when verifying one component. Several works have manipulated this technique notably in [8] where Grumberg and Long described the methodology using a subset of CTL in their framework and later in [9] where Henzinger and al. presented their successful implementations and case study regarding this approach.

Xie and Browne have proposed a method for software verification based on composition of several components [10]. Their main objective is developing components that could be reused with certitude that their behaviors will always respect their specification when associated in a proper composition. Therefore,

temporal properties of the software are specified, verified and packaged with the component for possible reuse. The implementation of this approach on software has been successful and the application of the assume-guarantee reasoning has considerably reduced the model checking complexity. A comprehensive approach to model-check component-based systems with abstraction refinement technique that uses verified properties as abstractions has been presented in [11].

In [12], Peng, Mokhtari and Tahar have presented a possible implementation of assume-guarantee approach where the specifications are in ACTL. Moreover, they managed to perform the synthetisation of the ACTL formulas into Verilog HDL behavior level program. The synthesized program can be used to check properties that the system's components must guarantee. Since, there have been other works on construction of components from interval temporal logic properties which could be used to speed up verification process [13, 14].

In 2007, a method to build abstractions of components into AKS (Abstract Kripke Structure), based on the set of the properties (CTL) each component verifies was presented in [15]. The method is actually a tentative to associate compositional and abstraction-refinement verification techniques. The generations of AKS from CTL formula have been successfully automated [16]. This work will be the base of the techniques in this paper.

Contribution: In this paper we present a strategy to exploit the properties of verified component in the goal of verifying complex systems with a good initial abstraction and eventually being conclusive in a small number of refinement iterations. We propose a technique to classify component properties according to their pertinency towards the global property, thus, enabling a better selection of properties for the initial abstraction generation. Furthermore, in the case where the verification is not conclusive, we propose a technique guided by the counterexample given by the model-checker to select supplementary properties to improve the abstraction.

In the next section, we will give an overview of our framework and introduce the notations that will be used later. The rest of the paper is organized as follows: Sect. 2.3 details our strategy of refinement. Section 2.4 presents the experimentation results and finally, Sect. 2.5 draws the conclusions and summarize our possible future works.

2.2 Our Framework

The model-checking technique we propose is based on the Counterexample-guided Abstraction Refinement (CEGAR) methodology [3]. The overall description of our methodology is shown in Fig. 2.1. We take into account the structure of the system as a set of synchronous components, each of which has been previously verified and a set of CTL properties is attached to each component. This set

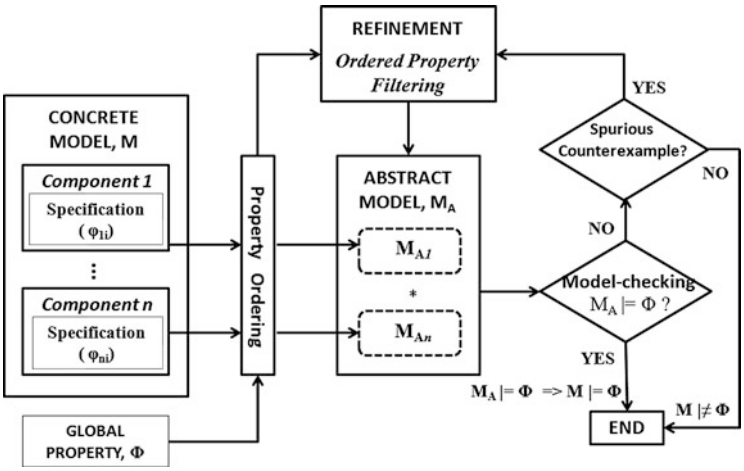

Fig. 2.1 Verification process

refers to the specification of the component. We would like to verify whether a concrete model, M presumably big sized and composed of several components, satisfies a global ACTL property Φ. Instead of building the product of the concrete components, we replace each concrete component by an abstraction of its behavior derived from a subset of the CTL properties it satisfies. Each abstract component represents an over-approximation of the set of behaviors of its related concrete component [15].

As shown in [17] for over-approximation abstraction, if Φ holds in the abstract model then it holds in the concrete model as well. However, if Φ does not hold in the abstract model then one cannot conclude anything regarding the concrete model until the counterexample has been analyzed. The test of spurious counter-example is then translated into a SAT problem as in [3]. When a counterexample is proven to be spurious, the refinement phase occurs, injecting more preciseness into the (abstract) model to be analyzed.

2.2.1 Concrete System Definition

As mentioned earlier, our concrete model consists of several components and each component comes with its specification. The concrete system is a synchronous composition of components, each of which described as a Moore machine.

Definition 2.1. A *Moore machine* C is defined by a tuple $\langle I, O, R, \delta, \lambda, \mathbf{R}_0 \rangle$, where,

- I is a finite set of Boolean input signals.
- O is a finite set of Boolean output signals.

- R is a finite set of Boolean sequential elements (registers).
- $\delta : 2^I \times 2^R \rightarrow 2^R$ is the transition function.
- $\lambda : 2^R \rightarrow 2^O$ is the output function.
- $\mathbf{R}_0 \subseteq 2^R$ is the set of initial states.

States (or configurations) of the circuit correspond to Boolean configurations of all the sequential elements.

Definition 2.2. A *Concrete system M* is obtained by synchronous composition of the component.
$M = C_1 \parallel C_2 \parallel \ldots \parallel C_n$,where each C_i is a Moore machine with a specification associated $\varphi_i = \{\varphi_i^1 \ldots \varphi_i^k\}$. Each φ_i^j is a CTL\X formula whose propositions AP belong to $\{I_i \cup O_i \cup R_i\}$.

2.2.2 Abstraction Definition

Our abstraction reduces the size of the representation model by letting free some of its variables. The point is to determine the good set of variable to be freed and when to free them. We take advantage of the CTL specification of each component: a CTL property may be seen as a partial view of the tree of behaviors of its variables configuration. All the variables not specified by the property can be freed. We introduced the Abstract Kripke Structure (AKS for short) which exactly specifies when the variable of the property can be freed. The abstraction of a component is represented by an AKS, derived from a subset of the CTL properties the component satisfies. Roughly speaking, AKS(φ), the AKS derived from a CTL property φ, simulates all execution trees whose initial state satisfies φ. In AKS(φ), states are tagged with the truth values of φ's atomic propositions, among the four truth values of Belnap's logic [18]: inconsistent ($\bot$), false (**f**), true (**t**) and unknown ($\top$). States with inconsistent truth values are not represented since they refer to non possible assignments of the atomic propositions. A set of fairness constraints eliminates non-progress cycles. The transformation algorithm of a CTL\X property into an AKS is described in [15, 19].

Definition 2.3. Given a CTL\X property φ whose set of atomic propositions is AP, an *Abstract Kripke Structure*, $AKS(\varphi) = (AP, \hat{S}, \hat{S}_0, \hat{L}, \hat{R}, \hat{F})$ is a 6-tuple consisting of:

- AP: The finite set of atomic propositions of property φ
- $\hat{S}$: a finite set of states
- $\hat{S}_0 \subseteq \hat{S}$: a set of initial states
- $\hat{L} : \hat{S} \rightarrow \mathscr{B}^{|AP|}$ with $\mathscr{B} = \{\bot, \mathbf{f}, \mathbf{t}, \top\}$: a labeling function which labels each state with configuration of current value of each atomic proposition.
- $\hat{R} \subseteq \hat{S} \times \hat{S}$: a transition relation where $\forall s \in \hat{S}, \exists s' \in \hat{S}$ such that $(s, s') \in \hat{R}$
- $\hat{F}$: a set of fairness constraints (generalized Büchi acceptance condition)

We denote by $\hat{L}(s)$, the configuration of atomic propositions in state s, and by $\hat{L}(s)[p]$, the projection of configuration $\hat{L}(s)$ according to atomic proposition p.

As the abstract model $\hat{M}$ is generated from the conjunction of verified properties of the components in the concrete model M, it can be seen as the composition of the AKS of each property. The AKS composition has been defined in [19]; it extends the classical synchronous composition of Moore machine to deal with four-valued variables.

Definition 2.4. An *Abstract model* $\hat{M}$ is obtained by synchronous composition of components abstractions. Let n be the number of components in the model and m be the number of selected verified properties of a component; let C_j be a component of the concrete model M and φ_j^k is a CTL formula describing a satisfied property of component C_j. Let $AKS(\varphi_{C_j^k})$ the AKS generated from φ_j^k. We have $\forall j \in [1,n]$ and $\forall k \in [1,m]$:

- $\hat{C}_j = AKS(\varphi_{C_j^1}) \,||\, AKS(\varphi_{C_j^2}) \,||\ldots||\, AKS(\varphi_{C_j^k}) \,||\ldots\, ||\, AKS(\varphi_{C_j^m})$
- $\hat{M} = \hat{C}_1 \,||\, \hat{C}_2 \,||\, \ldots \,||\, \hat{C}_j \,||\, \ldots \,||\, \hat{C}_n$

In an AKS, a state where a variable p is *unknown* can simulate all states in which p is either true or false. It is a concise representation of the set of more concrete states in which p is either true or false. A state s is said to be an *abstract state* if one of its variable p is *unknown*.

Definition 2.5. The *concretization* of an abstract state s with respect to the variable p (*unknown* in that state), assigns either true or false to p.
The *abstraction* of a state s with respect to the variable p (either true or false in that state), assigns *unknown* to p.

Property 2.1 (Concretization). Let A_i and A_j two abstractions such that A_j is obtained by concretizing one abstract variable of A_i (resp. A_i is obtained by abstracting one variable in A_j). Then A_i simulates A_j, denoted by $A_j \sqsubseteq A_i$.

Proof. As the concretization of state reduces the set of concrete configuration the abstract state represents but does not affect the transition relation of the AKS. The unroll execution tree of A_j is a sub-tree of the one of A_i. Then A_i simulates A_j. □

Property 2.2 (Composition and Concretization). Let $\hat{M}_i$ be an abstract model of M and φ_j^k be a property of a component C_j of M, $\hat{M}_{i+1} = \hat{M}_i \,||\, AKS(\varphi_j^k)$ is more concrete that $\hat{M}_i$, $\hat{M}_{i+1} \sqsubseteq \hat{M}_i$.

Proof. Let $s = (s_i, s_{\varphi_j^k})$ be a state in S_{i+1}, such that $s_i \in S_i$ and $s_{\varphi_j^k} \in S_{\varphi_j^k}$. The label of s_{i+1} is obtained by applying the Belnap's logic operators *and* to the four-valued values of variables in s_i and $s_{\varphi_j^k}$. For all $p \in AP_i \cup AP_{\varphi_j^k}$ we have the following label :

- $\hat{L}_{i+1}[p] = \top$ iff p is *unknown* in both states or does not belong to the set of atomic proposition.

- $\hat{L}_{i+1}[p] = \mathbf{t}$ (or $\mathbf{f}$) iff p is true (or false) in $s_{\varphi_j^k}$ (resp. s_i) and *unknown* in s_i (resp. $s_{\varphi_j^k}$).

By Property 2.1, $\hat{M}_{i+1}$ is more concrete than $\hat{M}_i$ and by the property of parallel composition, $\hat{M}_i \sqsubseteq \hat{M}_i \parallel AKS(\varphi_j^k)$. □

2.2.3 Initial Abstraction

Given a global property Φ, the property to be verified by the composition of the concrete components model, an abstract model is generated by selecting some of the properties of the components which are relevant to Φ. In the initial abstraction generation, all variables that appear in Φ have to be represented. Therefore the properties in the specification of each component where these variables are present will be used to generate the initial abstraction, $\hat{M}_0$ and we will verify the satisfiability of the global property Φ on this abstract model. If the model-checking failed and the counterexample given is found to be spurious, we will then proceed with the refinement process.

2.3 Refinement

2.3.1 Properties of Good Refinement

When a counterexample is found to be spurious, it means that the current abstract model $\hat{M}_i$ is too coarse and has to be refined. In this section, we will discuss about the refinement technique based on the integration of more verified properties of the concrete model's components in the abstract model to be generated. Moreover, the refinement step from $\hat{M}_i$ to $\hat{M}_{i+1}$ respects the properties below:

Definition 2.6. An efficient *refinement* verifies the following properties:

1. The new refinement is an over-approximation of the concrete model: $\hat{M} \sqsubseteq \hat{M}_{i+1}$.
2. The new refinement is more concrete than the previous one: $\hat{M}_{i+1} \sqsubseteq \hat{M}_i$.
3. The spurious counterexample in $\hat{M}_i$ is removed from $\hat{M}_{i+1}$.

Furthermore, the refinement steps should be easy to compute and ensure a fast convergence by minimizing the number of iterations of the CEGAR loop.

Refinements based on the concretization of selected abstract variables in $\hat{M}_i$ ensure Item 2. Concretization can be performed by modifying the AKS of $\hat{M}_i$ by changing some abstract value to concrete ones. However, this approach is rude: in order to ensure Item 1, the concretization needs to be consistent with the sequences of values in the concrete system. The difficulty resides in defining the proper abstract variable to concretize, at which precise instant, and with which Boolean value.

We propose to compose the abstraction with another AKS to build a good refinement according to Definition 2.6. We have several options. The most straightforward method consists in building an AKS representing all possible executions except the spurious counterexample; however the AKS representation may be huge and the process is not guaranteed to converge. A second possibility is to build an AKS with additional CTL properties of the components; the AKS remains small but Item 3 is not guaranteed, hence delaying the convergence. The final proposal combines both previous ones: first local CTL properties eliminating the spurious counterexample are determined, and then the corresponding AKS is synchronized with the one of $\hat{M}_i$.

2.3.2 Negation of the Counterexample

The counterexample at a refinement step i, σ, is a path in the abstract model $\hat{M}_i$ which dissatisfies Φ. In the counterexample given by the model-checker, the variable configuration in each state is Boolean. We name $\hat{L}_i$ this new labeling. The spurious counterexample σ is defined such that:

Definition 2.7. Let σ be a *spurious counterexample* in $\hat{M}_i = \langle AP_i, \hat{S}_i, \hat{S}_{0i}, \hat{L}_i, \hat{R}_i, \hat{F}_i \rangle$ of length $|\sigma| = n$: $\sigma = s_0 \rightarrow s_1 \ldots \rightarrow s_n$ with $(s_k, s_{k+1}) \in \hat{R}_i \; \forall k \in [0..n-1]$.

- All its variables are concrete: $\forall s_i$ and $\forall p \in AP_i$, p is either true or false according to $\hat{L}_i$. (not *unknown*), and s_0 is an initial state of the concrete system: $s_0 \in \mathbf{R}_0$
- σ is a counterexample in $\hat{M}_i$: $s_0 \not\models \Phi$.
- σ is not a path of the concrete system M: $\exists k \in [1..n-1]$ such that $\forall j < k, (s_j, s_{j+1}) \in R$ and $(s_k, s_{k+1}) \notin R$.

The construction of the AKS representing all executions except the one described by the spurious counterexample is done in two steps.

2.3.2.1 Step 1: Build the Structure of the AKS

Definition 2.8. Let σ be a spurious counterexampleof length $|\sigma| = n$, the *AKS of the counterexample negation* $AKS(\overline{\sigma}) = \langle AP_{\overline{\sigma}}, \hat{S}_{\overline{\sigma}}, \hat{S}_{0\overline{\sigma}}, \hat{L}_{\overline{\sigma}}, \hat{R}_{\overline{\sigma}}, \hat{F}_{\overline{\sigma}} \rangle$ is such that:

- $AP_{\overline{\sigma}} = AP_i$: The set of atomic propositions coincides with the one of σ
- $\hat{S}_{\overline{\sigma}}$: $\{s_T\} \cup \{s'_i | \forall i \in [0..n-2] \wedge s_i \in \sigma\} \cup \{\bar{s}_i | \forall i \in [0..n-1] \wedge s_i \in \sigma\}$
- $\hat{L}_{\overline{\sigma}}$ with $L_{\overline{\sigma}}(s'_i) = L_i(s_i), \forall i \in [0..n-2]$ and $L(s_T) = \{\top, \forall p \in AP_{\overline{\sigma}}\}$, $L_{\overline{\sigma}}(\bar{s}_i)$ is explained in the next construction step.
- $\hat{S}_{0\overline{\sigma}} = \{s'_0, \bar{s}_0\}$
- $\hat{R}_{\overline{\sigma}} = \{(\bar{s}_i, s_T), \forall i \in [0..n-1]\} \cup \{(s'_i, \bar{s}_{i+1}), \forall i \in [0..n-2]\} \cup \{(s'_i, s'_{i+1}, \forall i \in [0..n-3]\}$
- $\hat{F}_{\overline{\sigma}} = \emptyset$

The labeling function of s'_i represents (concrete) configuration of state s_i and state $\bar{s}_i$ represents all configurations *but* the one of s_i. This last set may not be

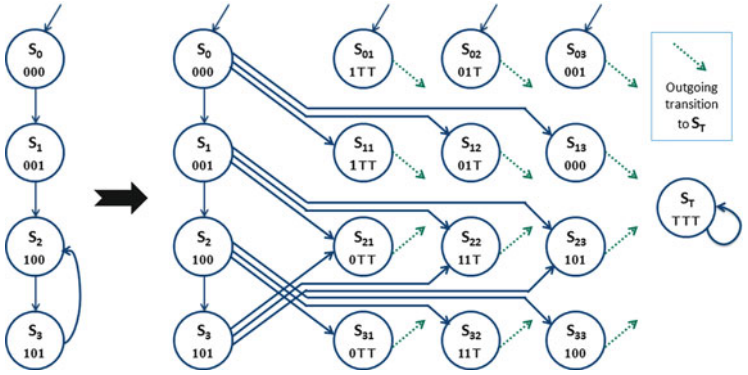

Fig. 2.2 An example of a negation of the counterexample AKS, $AKS(\sigma)$

representable by the labeling function defined in Definition 2.3. State labeling is treated in the second step. s_T is a state where all atomic propositions are *unknown*.

2.3.2.2 Step 2: Expand State Configurations Representing the Negation of a Concrete Configuration

The set of configurations associated with a state $\bar{s}_i$ represents the negation of the one represented by $L_i(s_i)$. This negation is not representable by the label of a single state but rather by a union of $\mid AP \mid$ labels.

Example: Assume $AP = \{v_0, v_1, v_2\}$ and $\sigma = s_0 \rightarrow s_1$ and $\hat{L}(s_0) = \{\mathbf{f}, \mathbf{f}, \mathbf{f}\}$ the configuration associated with s_0 assigns false to each variable. The negation of this configuration represents a set of seven concrete configurations which are covered by three (abstract) configurations: $\{\{\mathbf{t}, \top, \top\}, \{\mathbf{f}, \mathbf{t}, \top\}, \{\mathbf{f}, \mathbf{f}, \mathbf{t}\}\}$.

To build the final AKS representing all sequences but spurious counterexample σ, one replaces in $AKS(\overline{\sigma})$ each state $\bar{s}_i$ by $k = \mid AP_{\overline{\sigma}} \mid$ states $\bar{s}_i^j$ with $j \in [0..k-1]$ and assigns to each of them a label of k variables $\{v_0, \ldots, v_{k-1}\}$ defined such that: $\hat{L}(\bar{s}_i^j) = \{\forall l \in [0..j-1], v_l = L_i(s_i)[v_l]; v_j = \neg L_i(s_i)[v_j]; \forall l \in [j+1..k-1], v_l = \top\}$. Each state $\bar{s}_i^j$ is connected to the same predecessor and successor states as state $\bar{s}_i$. This final AKS presents a number of states in $\mathcal{O}(\mid \sigma \mid \times \mid AP \mid)$.

Figure 2.2 shows an example of the negation of a counterexample AKS built from a counterexample $\sigma = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_2$. The counterexample consists of four states with a loop to a previous state. The negation of the counterexample AKS allows all possible behaviors except the last step in σ. Therefore, the complementary states of every state in the counterexample are presented and at any step, a state in σ can proceed to these complementary states. The elimination of the last step is obtained by forcing its predecessor to the complementary states of the last step. All complementary states then leads to the terminal state, S_T which represents all possible behaviors in the future steps.

2.3.2.3 Reduction of the Negation of the Counterexample AKS

In the AKS generated, the set of configurations associated to the negation of a counterexample state may be redundant i.e. some configurations are represented several times in the AKS. Furthermore, all the states in negation part of the AKS have a unique successor namely the S_T state. Therefore, in the objective to reduce the number of states, these counterexample negation states with identical variable configurations can be merged. The merge definitions to generate the negation of the counterexample reduced AKS are given below.

Definition 2.9. *Merge condition:* Let $AKS(\bar{\sigma}) = (AP_{\bar{\sigma}}, \hat{S}_{\bar{\sigma}}, \hat{S}_{0\bar{\sigma}}, \hat{L}_{\bar{\sigma}}, \hat{R}_{\bar{\sigma}}, \hat{F}_{\bar{\sigma}})$. s_1 and s_2 are two counterexample negation states in M: $(s_1, s_2) \in \hat{S}_{\bar{\sigma}} \setminus \{s_T, s \in \sigma\}$. s_1 and s_2 can be merged iff

$$\hat{L}_{\bar{\sigma}}(s_1) = \hat{L}_{\bar{\sigma}}(s_2)$$

Definition 2.10. *Merging action:* Let $AKS(\bar{\sigma}) = (AP_{\bar{\sigma}}, \hat{S}_{\bar{\sigma}}, \hat{S}_{0\bar{\sigma}}, \hat{L}_{\bar{\sigma}}, \hat{R}_{\bar{\sigma}}, \hat{F}_{\bar{\sigma}})$ and its reduced AKS, $AKS(\bar{\sigma})' = (AP'_{\bar{\sigma}}, \hat{S}'_{\bar{\sigma}}, \hat{S}'_{0\bar{\sigma}}, \hat{L}'_{\bar{\sigma}}, \hat{R}'_{\bar{\sigma}}, \hat{F}'_{\bar{\sigma}})$ applying the Definition 2.9.

$s' \in \hat{S}', \forall (s_1, s_2) \in \hat{S} \setminus \{s_T, s \in \sigma\}$, $s' = \text{merge}\ (s_1, s_2) \Rightarrow$

- $\hat{L}'_{\bar{\sigma}}(s') = \hat{L}_{\bar{\sigma}}(s_1) = \hat{L}_{\bar{\sigma}}(s_2)$
- $\forall((s_{p1}, s_1), (s_{p2}, s_2)) \in \hat{R}^2, ((s_{p1}, s'), (s_{p2}, s')) \in \hat{R}'^2$
- $\forall((s_1, s_{s1}), (s_2, s_{s2})) \in \hat{R}^2, ((s', s_{s1}), (s', s_{s2})) \in \hat{R}'^2$

Property 2.3. $AKS(\bar{\sigma})'$ and $AKS(\bar{\sigma})$ are bisimulation-equivalent:

$$AKS(\bar{\sigma})' \sim AKS(\bar{\sigma})$$

Proof. Let $AKS(\bar{\sigma}) = (AP_{\bar{\sigma}}, \hat{S}_{\bar{\sigma}}, \hat{S}_{0\bar{\sigma}}, \hat{L}_{\bar{\sigma}}, \hat{R}_{\bar{\sigma}}, \hat{F}_{\bar{\sigma}})$ and its reduced AKS, $AKS(\bar{\sigma})' = (AP'_{\bar{\sigma}}, \hat{S}'_{\bar{\sigma}}, \hat{S}'_{0\bar{\sigma}}, \hat{L}'_{\bar{\sigma}}, \hat{R}'_{\bar{\sigma}}, \hat{F}'_{\bar{\sigma}})$.
All the initial states in $\hat{S}_{0\bar{\sigma}}$ are represented in $\hat{S}'_{0\bar{\sigma}}$ and vice versa. $\forall (s_1, s_2) \in \hat{R}_{\bar{\sigma}}, \exists (s'_1, s'_2) \in \hat{R}'_{\bar{\sigma}}$ where $\hat{L}_{\bar{\sigma}}(s_i) = \hat{L}'_{\bar{\sigma}}(s'_i)$, and the other way around is also true.
Therefore, $AKS(\bar{\sigma})' \sim AKS(\bar{\sigma})$. □

Figure 2.3 demonstrates the gain from the reduction process on the generation of the negation of the counterexample AKS from the counterexample σ in the previous example. In the Fig. 2.3 above, we can see that all the complementary states have a unique variable configuration and the duplicates no longer present in the AKS. This simplification technique helps to reduce the size of the system without having a degradation in terms of property verification as the resulted AKS is bisimilar to the original one. Even though the gain may seem insignificant at first sight, the reduction done may be precious when the technique in conducted on many refinement iterations. Therefore, this reduction technique will be applied systematically on this method of refinement.

However, removing, at each refinement step, the spurious counterexample *only* induces a low convergence. Moreover, in some cases, this strategy may not converge: suppose that all sequences of the form $a.b^*.c$ are spurious counterexamples (here a, b and c represent concrete state configurations). Assume, at a given

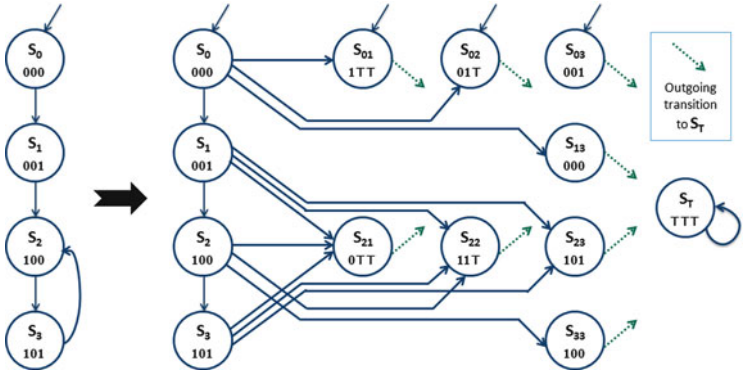

Fig. 2.3 An example of a reduced negation of the counterexample AKS, $AKS(\sigma)'$

refinement step i, a particular counterexample $\sigma_i = s_0 \rightarrow s_1 \rightarrow \ldots s_n$ with $L(s_0) = a, \forall k \in [1, n-1], L(s_k) = b, L(s_n) = c$. Removing this counterexample does not prevent from a new spurious counterexample at step $i+1$: $\sigma_{i+1} = s_0 \rightarrow s_1 \rightarrow \ldots s_{n+1}$ with $L(s_0) = a, \forall k \in [1, n], L(s_k) = b, L(s_{n+1}) = c$. The strategy consisting of elimination spurious counterexample *one by one* diverges in this case. Furthermore, we cannot eliminate all the sequences of the form $a.b^*.c$ in a unique refinement step since we do not a priori know if at least one of these sequences is executable in the concrete model.

Therefore, from these considerations, we are interested in removing *sets of behaviors encompassing the spurious counterexample* while still guaranteeing an over-approximation of the set of tree-organized behaviors of the concrete model. The strengthening of the abstraction $\hat{M}_i$ with the addition of AKS of already verified local CTL properties eliminates sets of behaviors and guarantees the over-approximation (Property 2.2) but does not guarantee the elimination of the counterexample. We present in the following section a strategy to select sets of CTL properties eliminating the spurious counterexample.

2.3.3 Ordering of Properties

We propose a heuristic to order the properties depending on the structure of each component. In order to do so, the variable dependency of the variables present in global property has to be analyzed. After this point, we refer to the variables present in the global property as *primary variables*.

We observed that the closer a variable is to the primary variable, the higher influence it has on it. Moreover, a global property often specifies the behavior at the interface of components. Typically, a global property ensures that a message sent is always acknowledged or the good target gets the message. This kind of behavior relates the input-output behaviors of components. We have decided to allocate an

extra weight for interface variables whereas variables which do not interfere with a primary variable are weighted 0. Here is how we proceed:

1. Build the structural dependency graph for all primary variables.
2. Compute the depth of all variables in all dependency graphs. Note that a variable may belong to more than one dependency graph, in that case we consider the minimum depth.
3. Give a weight to each variable (see Algorithm 1).
4. Compute the weight of properties for each component: sum of the property variables weight.

Algorithm 1: Compute weight

Input: G, the set of all dependency graph variable
V, the set of variables
Output: $\{(v,w)|v \in V, w \in \mathbb{N}\}$, The set of variables with their weight

begin
  $p = \max(\text{depth(G)})$
  for $v \in V$ **do**
    $d = \text{depth}(v)$;
    $w = 2^{p-d} * p$;
    if $d == 0$ **then** v is primary variable
      $w = 5 * w$;
    end
    if $v \in I \cup O$ **then** v is an interface variable
      $w = 3 * w$
    end
  end
end

The Algorithm 1 gives weight according to the variable distance to the primary variable with extra weight for interface variable and primary variable. It is definitely not an exact pertinence calculation of properties but provides a good indicator of their possible impact on the global property. After this pre-processing phase, we have a list of properties ordered according to their pertinence with regards to the global property.

2.3.3.1 Example

In this example, we have a global property $\phi = A((p = 1)U(q = 1))$; which consists of two primary variables: p and q. As shown in Fig. 2.4, the primary variable p is dependent of three the other variables: x, y and z whereas the primary variable q is dependent of four variables: r, s, u and v. The maximum depth of between the two primary variables dependency graphs is three ($q \leftarrow r \leftarrow u \leftarrow v$). Furthermore, apart from p and q being the primary variables, we have y, z, s and v which are interface

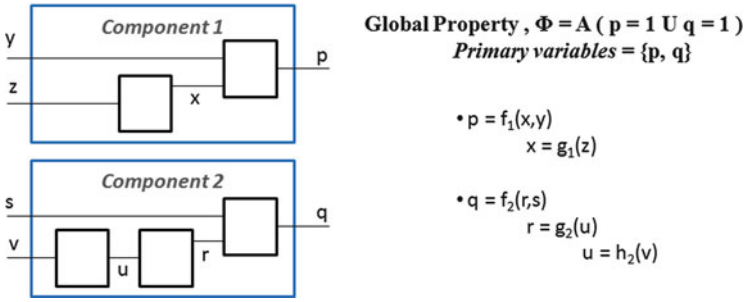

Fig. 2.4 Example of variable dependency

variables. Let's assume that we have a set of properties that includes $\varphi_a - \varphi_g$, with the weight computation algorithm given previously, the property φ_a which consists of variables p, y and z will therefore obtain the highest total weight and the rest of the properties will be ordered as follows:

List of ordered component properties:

1. $\varphi_a(p, y, z)$
2. $\varphi_b(q, s, v)$
3. $\varphi_c(p, y)$
4. $\varphi_d(q, r, v)$
5. $\varphi_e(p, z)$
6. $\varphi_f(x, z)$
7. $\varphi_g(r, u, v)$
8. ...

Here we can see that the top property φ_a only consists of primary variable p, therefore the highest property in the list containing q will also be selected in the initial abstraction generation.

Selected properties for the initial abstraction:

1. $\varphi_a(p, y, z)$
2. $\varphi_b(q, s, v)$

2.3.4 Filtering Properties

The refinement step consists of adding new AKS of properties selected according to their pertinence. As we would like to ensure the elimination of the counterexample previously found, we filter out properties that do not have an impact on the counterexample σ thus will not eliminate it. In order to reach this objective, a Abstract Kripke structure of the counterexample σ, $K(\sigma)$ is generated. $K(\sigma)$ is a

succession of states corresponding to the counterexample path which dissatisfies the global property Φ.

Definition 2.11. Let σ be a counterexample of length n in $\hat{M}_i$ such that $\sigma = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_{n-1}$. The *Kripke structure derived from* σ is 6-tuple $K(\sigma_i) = (AP_\sigma, S_\sigma, S_{0\sigma}, L_\sigma, R_\sigma, F_\sigma)$ such that:

- $AP_\sigma = AP_i$: a finite set of atomic propositions which corresponds to the variables in the abstract model
- $S_\sigma = \{s_i | s_i \in \sigma\} \cup \{s_T\}$
- $S_{0\sigma} = \{s_0\}$
- $L_\sigma = \hat{L}_i \cup L(s_T) = \{\top, \forall p \in AP_\sigma\}$
- $R_\sigma = \{(s_k, s_{k+1}) | (s_k \rightarrow s_{k+1}) \in \sigma\} \cup \{(s_{n-1}, s_T)\}$
- $F_\sigma = \emptyset$

All the properties available for refinement are then model-checked on $K(\sigma)$. If the property holds then the property will not eliminate the counterexample. Hence this property is not a good candidate for refinement. Therefore the highest weighted property not satisfied in $K(\sigma)$ is chosen to be integrated in the next refinement step. This process is iterated for each refinement step.

Property 2.4. **Counterexample eviction**

1. If $K(\sigma) \models \varphi \Rightarrow AKS(\varphi)$ will not eliminate σ.
2. If $K(\sigma) \not\models \varphi \Rightarrow AKS(\varphi)$ will eliminate σ.

Proof. 1. By construction, $AKS(\varphi)$ simulates all models that verify φ. Thus the tree described by $K(\sigma)$ is simulated by $AKS(\varphi)$, it implies that σ is still a possible path in $AKS(\varphi)$.
2. $K(\sigma)$, where φ does not hold, is not simulated by $AKS(\varphi)$, thus σ is not a possible path in $AKS(\varphi)$ otherwise $AKS(\varphi) \not\models \varphi$ that is not feasible due to AKS definition and the composition with M_i with $AKS(\varphi)$ will eliminate σ. □

The proposed approach ensures that the refinement excludes the counterexample and respects the Definition 2.6. We will show in our experiments that first, the time needed to build an AKS is negligible and secondly the refinement converges rapidly.

2.4 Experimental Results

We have conducted preliminary experiments to test and compare the performance of our strategy with existing techniques available in VIS. There are several abstraction-refinement techniques implemented in VIS accessible via *approximate_model_check*, *iterative_model_check*, *check_invariant* and *incremental_ctl_verification* commands. However, among the available techniques, *incremental_ctl_verification* is the only one that supports CTL formulas and fairness constraints which are necessary in our test platforms. It is an automatic abstraction

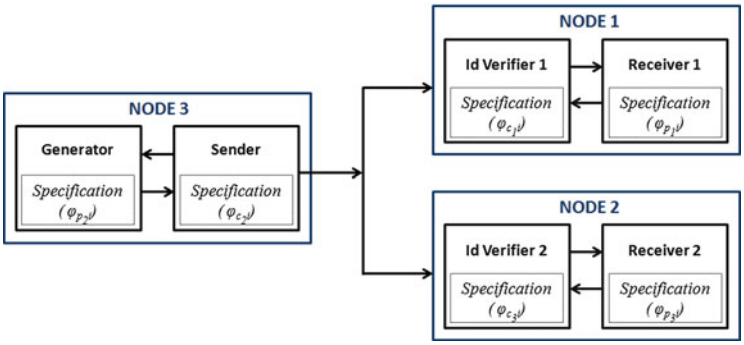

Fig. 2.5 CAN protocol platform

Table 2.1 Statistics on the VCI-PI and CAN bus platform

Experiment platform			Number of BDD variables	BDD size	Number of reachable states	Analysis time (s)
		1 master-1 slave	304	7,207	4.711e + 3	6.36
	Concrete	2 masters-1 slave	445	24,406	7.71723e + 06	35.2
	model	4 masters-1 slave	721	84,118	3.17332e + 12	2,818.3
		4 masters-2 slaves	895	238,990	5.708e + 15	68,882.3[a]
VCI-PI	Final	1 master-1 slave	197	76	5.03316e + 07	0.01
	abstract	2 masters-1 slave	301	99	4.12317e + 11	0.02
	model	4 masters-1 slave	501	147	3.45876e + 18	0.03
	for ϕ_1	4 masters-2 slaves	589	167	7.08355e + 21	0.04
	Final	1 master-1 slave	194	50	2.62144e + 07	0
	abstract	2 masters-1 slave	298	73	2.14748e + 11	0.01
	model	4 masters-1 slave	498	121	1.80144e + 18	0.02
	for ϕ_2	4 masters-2 slaves	586	141	3.68935e + 21	0.02
CAN bus	Concrete model		822	161,730	3.7354e + 07	300.12
	Final abstract model for ϕ_3		425	187	1.66005e + 12	0.03
	Final abstract model for ϕ_4		425	187	1.66005e + 12	0.04

[a] Computed on a calculation server: 2x Xeon X5650, 72 Go RAM

refinement algorithm which generates an initial conservative abstraction principally by reducing the size of the latches by a constant factor. If the initial abstraction is not conclusive, a *goal set* will then be computed in order to guide the refinement process [20, 21].

We have executed and compared the execution time and the number of refinement iterations for two examples: VCI-PI platform consisting of Virtual Component Interface (VCI), a PI-Bus and VCI-PI protocol converter and a simplified version of a CAN bus platform consisting of three nodes on a CAN bus as shown in Fig. 2.5. Table 2.1 gives the size and the statistics concerning the VCI-PI platform and CAN bus platform verified. All the values are obtained using the *compute_reach*

Table 2.2 Verification Results

Experiment platform	Global property	Verification technique	Refinement iteration	Verification time (s)
		Property selection	1	2.2
VCI-PI:	ϕ_1	Incremental	0	6.3
1 master		Standard MC	–	6.06
–		Property selection	0	1.0
1 slave	ϕ_2	Incremental	562	200.9
		Standard MC	–	6.13
		Property selection	1	2.0
VCI-PI:	ϕ_1	Incremental	0	20.4
2 masters		Standard MC	–	37.9
–		Property selection	0	1.0
1 slave	ϕ_2	Incremental	74	786.3
		Standard MC	–	39.4
		Property selection	1	2.1
VCI-PI:	ϕ_1	Incremental	0	261.6
4 masters		Standard MC	–	>1 day
–		Property selection	0	1.0
1 slave	ϕ_2	Incremental	0	263.5
		Standard MC	–	>1 day
		Property selection	1	2.2
VCI-PI:	ϕ_1	Incremental	N/A	>1 day
4 masters		Standard MC	–	>1 day
–		Property selection	0	1.1
2 slaves	ϕ_2	Incremental	N/A	>1 day
		Standard MC	–	>1 day
		Property selection	0	1.02
	ϕ_3	Incremental	N/A	>1 day
CAN		Standard MC	–	2,645.4
bus		Property selection	0	1.01
	ϕ_4	Incremental	N/A	>1 day
		Standard MC	–	1,678.1

command with option *-v 1* in VIS except the number of BDD variables, computed using the *print_bdd_stats* command. The experiments have been executed on a PC with an AMD Athlon dual-core processor 4450e and 1.8 GB of RAM memory.

In Table 2.2, we compare the execution time and the number of refinement between our technique (Prop. Select.), *incremental_ctl_verification* (Incremental) and the standard model checking (Standard MC) computed using the *model_check* command in VIS (Note: Dynamic variable ordering has been enabled with sift method). For the VCI-PI platform, the global property ϕ_1 is the type $AF((p = 1) * AF(q = 1))$ and ϕ_2 is actually a stronger version of the same formula with $AG(AF((p = 1) * AF(q = 1)))$ where all requests to write on the PI-Bus will finally be granted in the future. We have a total of 26 verified components properties to be selected in the VCI-PI platform. In comparison to ϕ_2, we can see that, a better

set of properties available will result in a better abstraction and less refinement iterations.

In the case of the CAN bus platform, the global property ϕ_3 is the type $AG(((p' = 1) * (q' = 1) * AF(r_1 = 1)) \rightarrow AF((s_1 = 1) * AF(t_1 = 1)))$ and $\phi_4 = AG(((p' = 1) * (q' = 1) * AG(r_2 = 0)) \rightarrow AG((s_2 = 0) * (t_2 = 0)))$. They describe the correct transmission of generated messages to the receivers. We have at our disposal 103 verified component properties and after the selection process for the initial abstraction, 3 selected component properties were sufficient to verify both global properties without refinement.

Globally, we can see that our technique, for these examples, systematically computes faster than the other two methods and interestingly in the case where the size of the platform increases by adding more connected components, in contrary to the other two methods, our computation time remains stable. This is mainly due to the fact that for small number of properties, our abstraction is generated almost instantly and as only pertinent properties are selected, not many refinement iterations are required in order to complete the verification process. It is also important to note that the properties tested are simple and we have in our property selection list the local properties required to satisfy the global property.

2.5 Negation of the Counterexample as a Complementary Strategy

A well constituted specification is a prerequisite for an efficient refinement strategy based on property selection technique. However, in practice, we don't always have at our disposal a complete specification. Hence, it may be possible that at a particular refinement iteration, none of the properties available is capable of eliminating the counterexample. In this case, we propose the negation of the counterexample technique as a complementary strategy.

Let's suppose that there are no properties available to refine our CAN Bus abstract model for the verification of a global property $\phi_5 = A((a = 1)U((a = 0) * AX((b_1 = 1) * (b_2 = 1))))$; where b_1 and b_2 are outputs of the Receiver 1 and 2 respectively and they are set to 1 in the next step after the signal $a = 0$ is on the bus which indicates the start of frame. As our current AKS generator is only capable of generating $CTL \setminus X$ properties only, the initial abstractions of each component were built with the aid of the AF operators which allows more satisfaction configurations than the AX operator.

Therefore, in this example, the negation of the counterexample strategy could help to eliminate the different configurations that are present in the abstraction. The first counterexample σ_1 provided by the model checker gives the undesired configuration where the output b_1 still remains at 0 right after $a = 0$. Thus, the negation of the counterexample is applied on this counterexample configuration to eliminate it.

Table 2.3 Statistics at each refinement step with the negation of counterexample technique

Iteration	Number of BDD variables	BDD size	Number of reachable states	Model checking result
0	424	199	4.00015e + 09	$\widehat{M_0} \not\models \phi_5$
1	426	199	3.79804e + 09	$\widehat{M_1} \not\models \phi_5$
2	428	249	3.6633e + 09	$\widehat{M_2} \models \phi_5$

In the following iteration, the model-checker provides a rather similar configuration of undesired behavior with this time the output b_2 which remains at 0 after $a = 0$. As previously done, the negation of counterexample is applied on this counterexample σ_2. Finally, after these two refinement iterations, the abstract model built managed to satisfy the property ϕ_5. Table 2.3 shows some statistics at each refinement iteration.

2.6 Conclusion and Future Works

We have presented a new strategy in the abstraction generation and refinement which is well adapted for compositional embedded systems. This verification technique is compatible and suits well in the natural development process of complex systems. Our preliminary experimental results show an interesting performance in terms of duration of abstraction generation and the number of refinement iteration. Moreover, this technique enables us to overcome repetitive counterexamples due to the presence of cycles in the system's graph.

Nevertheless, in order to function well, this refinement technique requires a well constituted specification of every components of the concrete model. Furthermore, it may be possible that none of the properties available is capable of eliminating the counterexample which is probably due to an incomplete specification or a counterexample that should be eliminated by the product of local properties.

We have also demonstrated a possible application of the negation of the counterexample technique as a complementary strategy albeit limited to certain form of counterexamples only. Indeed, the negation of counterexample technique is inefficient when dealing with counterexample with a cycle in the prefix (e.g. $a.b^*.c$).

In this case, other refinement techniques such as the identification of a good set of local properties to be integrated simultaneously should be considered. We are currently investigating other complementary techniques to overcome these particular cases. The work of Kroening [22], for example, could also help us in improving the specification of the model: at the component level, or for groups of components.

Furthermore, we are also examining a comprehensive new strategy that exploits the finite-state machines (FSMs) of the components in the verification process. A procedure to generate properties which are directly derived from the

component's FSM structure is considered as a solution to overcome the insufficiency of component properties to be selected for the abstraction generation. These ongoing researches will enrich the existing verification techniques in property based abstraction generation.

Acknowledgements We thank Neha Agarwal for the implementation of the negation of the counterexample (without reduction) AKS generator which is the base of the generator with reduction used in this paper.

References

1. The VIS Group: VIS: A system for verification and synthesis, In: Alur, R., Henzinger, T.A. (eds.) Proceedings of the 8th International Conference, CAV '96, New Brunswick. LNCS, vol. 1102, pp. 428–432. Springer, Berlin/Heidelberg (1996)
2. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Computer Aided Verification (CAV '97), Haifa. LNCS, vol. 1254. Springer, London, Springer Berlin Heidelberg (1997)
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV'00, Chicago. LNCS (2000)
4. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In: 4th International Conference on Integrated Formal Methods, Canterbury, vol. 2999, pp. 1–20. Springer (2004)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast: Applications to software engineering. Int. J. Softw. Tools Technol. Trans. **9**(5–6), 505–525 (2007)
6. Jain, H., Kroening, D., Sharygina, N., Clarke, E.: VCEGAR: Verilog counterexample guided abstraction refinement. In: TACAS'07, Braga, 2007
7. Sharygina, N., Tonetta, S., Tsitovich, A.: An abstraction refinement approach combining precise and approximated techniques. Int. J. Softw. Tools Technol. Trans. **14**, 1–14 (2012)
8. Grumberg, O., Long, D.E.: Model checking and modular verification. In: International Conference on Concurency Theory, Amsterdam, vol. 527, pp. 250–263. Springer, Berlin/Heidelberg (1991)
9. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You assume, we guarantee: Methodology and case studies. In: CAV '98, Vancouver, vol. 1427, pp. 440–451. Springer, Berlin/Heidelberg (1998)
10. Xie, F., Browne, J.C.: Verified systems by composition from verified components, in ESEC/FSE 2003. In: 11th ACM SIGSOFT Symposium on Foundations of Software Engineering Conference, Helsinki, pp. 227–286. ACM (2003)
11. Li, J., Sun, X., Xie, F., Song, X.: Component-based abstraction refinement. In: 10th International Conference on Software Reuse (ICSR), Beijing, pp. 39–51. Springer (2008)
12. Peng, H., Mokhtari, Y., Tahar, S.: Environment synthesis for compositional model checking. In: ICCD '02: 20th International Conference on Computer Design, Freiburg, pp. 70–75. IEEE Computer Society (2002)
13. Schickel, M., Nimbler, V., Braun, M., Eveking, H.: On consistency and completeness of property-sets: Exploiting the property-based design process. In: FDL '06: Forum on Specification and Design Languages, Darmstadt (2006)
14. Nguyen, M.D., Wedler, M., Stoffel, D., Kunz, W.: Formal hardware/software co-verification by interval property checking with abstraction. In: Design Automation Conference (DAC'11), San Diego 2011

15. Braunstein, C., Encrenaz, E.: Using CTL formulae as component abstraction in a design verification flow. In: ACSD, Bratislava, pp. 80–89. IEEE Computer Society (2007)
16. Bara, A.: Abstraction de Composant pour la Vérification par Model-Checking, Mémoire de Diplôme Universitaire OMP – LIP6-SOC, (2008)
17. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
18. Belnap, N.: A useful four-valued logic. In: Modern Uses of Multiple-Valued Logic, pp. 8–37. Springer, Berlin/Heidelberg (1977)
19. Braunstein, C.: Conception Incrémentale, Vérification de Composants Matériels et Méthode d'abstraction pour la Vérification de Systèmes Intégrés sur Puce. Ph.D. thesis, Université Pierre et Marie Curie, LIP6-SOC (2007)
20. Pardo, S., Hachtel, G.: Incremental CTL model checking using BDD subsetting. In: DAC '98: 35th Design Automation Conference, San Francisco, pp. 457–462. ACM (1998)
21. Pardo, S., Hachtel, G.: Automatic abstraction technique for propositional mu-Calculus model checking. In: CAV '97, Haifa, vol. 1254, pp. 12–23. Springer, (1997)
22. Purandare, M., Wahl, T., Kroening, D.: Strengthening properties using abstraction refinement. In: Proceedings of DATE '09, Nice, pp. 1692–1697. ACM (2009)

http://www.springer.com/978-3-319-01417-3

Models, Methods, and Tools for Complex Chip Design
Selected Contributions from FDL 2012
Haase, J. (Ed.)
2014, XV, 221 p. 94 illus., 57 illus. in color., Hardcover
ISBN: 978-3-319-01417-3