J Supercomput (2012) 62:828–854
DOI 10.1007/s11227-012-0755-0

State space reduction in modeling checking parameterized cache coherence protocol by two-dimensional abstraction

Yang Guo · Wanxia Qu · Long Zhang · Weixia Xu

Published online: 17 April 2012

Abstract Scalability of cache coherence protocol is a key component in future shared-memory multi-core or multi-processor systems. The state space explosion is the first hurdle while applying model-checking to scalable protocols. In order to validate parameterized cache coherence protocols effectively, we present a new method of reducing the state space of parameterized systems, two-dimensional abstraction (TDA). Drawing inspiration from the design principle of parameterized systems, an abstract model of an unbounded system is constructed out of finite states. The mathematical principles underlying TDA is presented. Theoretical reasoning demonstrates that TDA is correct and sound. An example of parameterized cache coherence protocol based on MESI illustrates how to produce a much smaller abstract model by TDA. We also demonstrate the power of our method by applying it to various well-known classes of protocols. During the development of TH-1A supercomputer system, TDA was used to verify the coherence protocol in FT-1000 CPU and showed the potential advantages in reducing the verification complexity.

Y. Guo (✉) · L. Zhang
Institute of Microelectronics and Microprocessor, School of Computer Science, National University of Defense Technology, Changsha, P.R. China
e-mail: guoyang@nudt.edu.cn

L. Zhang
e-mail: saphira_long@qq.com

W. Qu · W. Xu
Institute of Computer, School of Computer Science, National University of Defence Technology, Changsha, P.R. China

W. Qu
e-mail: quwanxia@nudt.edu.cn

W. Xu
e-mail: xuwx@nudt.edu.cn

Keywords Parameterized cache coherence protocol · True concurrency · Model checking · Two-dimensional abstraction

1 Introduction

Model checking is an automatic technique for verifying finite state concurrent systems, which uses a finite state machine to describe the system under consideration and temporal logic to state the properties that the system must satisfy. This method has been used successfully in practice to verify complex software and hardware systems [1, 2]. However, efficient verification of parameterized cache coherence protocols is one of the most challenging problems in verification domain today. Firstly, parameterized systems are composed of an arbitrary number of processes which concur cooperatively (the number of processes is called the system parameter). The behavior of one process is determined not only by its current state, but also the changes of the environment it lives. Secondly, parameterized systems are by nature unbounded. The system parameter may be arbitrarily large, and the ultimate goal is to validate the properties in a system for every possible number of processes. In such cases, the number of global states can be enormous, resulting in the state space explosion. Formal verification of parameterized systems is known to be undecidable and thus cannot be automated. Thirdly, symbolic methods such as BDD or SAT, which can enable scalable formal verification methods, can be ineffective when it comes to cache coherence protocols because most of the state variables are relevant in protocol property verification. As faster larger systems are designed, the complexity of cache protocols will continue to increase.

Fong Pong [3] presented a comprehensive survey of various approaches to the verification of cache coherence protocol based on state enumeration, model checking, and symbolic state models. He pointed out that no framework had been proposed so far to deal with the memory consistency model in the context of formal verification based on state expansion. Monolithic formal verification methods that treat the protocol as a whole have been used fairly routinely for verifying cache coherence protocols from the early 1990s [4, 5]. However, these monolithic techniques will not be able handle the very large state space of parameterized protocols. While techniques like indexed predicates [6], counter abstraction [7], environment abstractions [8, 9], and cutoffs based approach [10] have been proposed for parameter protocol verification during these years, none of them scales well to large protocols, and those that do scale require an inordinate amount of manual effort to succeed [11]. We are not aware of any published work that has reported formal verification of a parameterized cache coherence protocol with reasonable complexity.

All successful applications of model checking thus far have made use of domain specification abstraction techniques. Continuing this trend and drawing inspiration from recent work like environment abstraction [8, 9], we exploit the domain knowledge about parameterized systems to devise an appropriate abstraction method. We propose a novel generic approach called two-dimensional abstraction (TDA), which could effectively reduce the state space of parameterized systems. In our work, the size of the state transition graph for each process is reduced independently at first, then the whole system composed of the reduced processes is abstracted based on the design principles of parameterized systems, thus avoiding the construction of the complete state space that might be too large to fit into memory.

TDA has a number of advantages over other approaches. First, TDA abstracts away redundant information from a concrete system via decomposition–abstraction–composition–reabstraction, thus effectively alleviating the state explosion problem during parameterized systems verification. Second, TDA can be used for parallel systems in the usual fashion because it has no limitation in communication mode among processes. Third, TDA can be used with any model checker. The freedom to choose model checkers is important in practice. Fourth, TDA is sound and complete. We give complete soundness and completeness proofs for our method. At last, constant heterogeneous processes and infinite state systems are allowed, which makes TDA suitable for large scale heterogeneous systems. We demonstrate the power of our method by applying it to various well-known classes of protocols.

The rest of this paper is organized as follows. In Sect. 2, we introduce previous related work. Section 3 gives some background information. In Sect. 4, we propose a model with true concurrency semantics for parameterized systems. In Sect. 5, we present concepts of a TDA model and the method to construct a TDA model. A cache coherence protocol based on MESI is used to illustrate the approach of getting a much smaller state space by TDA in Sect. 6. Experimental results of various well-known protocols and application are presented in Sect. 7. Section 8, the last section, presents concluding remarks.

2 Related works

The development of effective techniques for checking parameterized systems is one of the most challenging problems in verification today. Prior research in the area of coherence protocol verification has ranged from simulation to formal methods. These techniques have had varying degrees of success, but few of them have been applied to a large industrial-strength protocol like FLASH.

Simulation with random or directed stimulus has been shown to be effective at finding most protocol errors [12]. However, simulation tends not to be effective at uncovering subtle bugs, especially those related to the consistency model. Subtle consistency bugs often occur only under unusual combinations of circumstances, and it is unlikely that simulation will drive the protocol to these situations.

For verification of high level specifications, modern industrial practice consists of modeling small instances of the protocols in guard/action languages such as Murphi [13] or TLA+ [14], and exploring the reachable states through explicit state enumeration.

The idea of using non-interference lemmas for parameterized model checking is attributed to McMillan [15], Chou [16], and Li [17], which is also called the CMP method. The CMP approach to parameterized verification is a combination of data type reduction and compositional reasoning. In this approach, a model checker is used as proof assistant and the user guides the proof by supplying invariants or non-interference lemmas. Similar types of reasoning have been applied by Chen to verify non-parameterized hierarchical protocols [18]. The compositional method of McMillan is used for compositional reasoning to handle infinite state systems including directory based protocols. This technique, which requires user intervention at various stages, has been applied to verify safety and liveness properties of the FLASH protocol. The paper by Chou [16] presented a method along similar lines, that was used to verify safety of FLASH and GERMAN protocol. Krstic [19] gave a formalization of the method. The CMP method scales well. As far as we are aware, the CMP method is one of a few methods to handle the full complexity of the FLASH protocol. Intel used CMP to verify an industrial-strength cache protocol several orders of magnitude larger than even the FLASH protocol [20]. Talupur and Tuttle showed how to derive high-quality invariants from message flows and how to use these invariants to accelerate the CMP method [21, 22]. A message flow is a sequence of messages sent among processors during the execution of a protocol. The hardest part of using CMP is finding a set of protocol invariants that enable CMP to work. The user has the burden of coming up with non-interference lemmas which can be non-trivial and require deep understanding of the protocol under verification.

Another effective method for parameterized verification is the abstraction approach [6–9, 11, 23–25]. Predicate abstraction, first proposed by Graf [11] as a special case of the general framework of abstraction interpretation, has been used in the verification of parameterized protocols. In predicate abstraction, a finite set of predicates is defined over the concrete set of states. These predicates are used to construct a finite state abstraction of a concrete system. The automation in generating the finite abstract model makes this scheme attractive in combining deductive and algorithmic approaches for infinite state verification. Lahiri [26] proposed the use of a symbolic decision procedure and its application for predication abstraction. One of the main problems in predicate abstractions is that it typically makes a large number of theorem prover calls when computing the abstract transition relation or the abstract state space. Pnueli [23] presented the method of invisible invariants that combines a small-model theorem with a heuristics to generate proofs of correctness of parameterized systems. Wang [24] used monotonic abstraction to provide an over-approximation of the transition system induced by a parameterized system. The over-approximation gives a transition system which is monotonic with respect to a well quasi-ordering on the set of configurations. Timm [25] presented an approach combining symmetry arguments with spotlight abstractions. The technique determines (the size of) a particular instantiation of the parameterized system from the given temporal logic formula, and feds this into an abstracting model checker. Environment abstraction [8, 9] exploits the replicated structure of a parameterized system to make its verification easy, and it converts the unbounded system into a bounded one via finite state description method. In real cache coherence protocols, the internal state of each cache can be quite complex, and thus environment abstraction might fail. The other method is divide-and-conquer, in other words, abstraction for each process is made independently before the model for the whole system is constructed [27]. Unfortunately, too many constraints for systems under consideration make this way unpractical.

Other related work includes that of Pandav [28] who has proposed a set of heuristics to aid in constructing invariants for cache protocols. Delzanno [29] used arithmetic constraints to model possibly infinite sets of global states of a multi-processor system with many identical caches. General purpose symbolic model checkers for infinite-state systems working over arithmetical domains were used. Delzanno and Bultan [30, 31] described a constraint based verification method for handling the safety and liveness properties of GERMAN protocol. But their method cannot verify single index liveness properties. Emerson and Kahlon [32] verified GERMAN by first reducing it to a snoopy bus protocol and then invoking a theorem asserting that if a snoopy bus protocol of a certain form is correct for 7 nodes then it is correct for any number of nodes. Pnueli proposed an elegant cutoff method that can verify the DIR protocol [10], but it was sound and not complete, and worked only for safety properties. A broad technique was proposed for the verification of WSIS systems that can handle the DIR protocol as an example [33], yet again the resulting technique was sound but not complete.

3 Preliminaries

This section contains basic material about the Kripke structure, temporal logic and equivalent relation on Kripke structures [34].

Definition 1 (Kripke structure) Let AP be a set of atomic propositions. A *Kripke structure* M over AP is a five-tuple $M = (AP, S, I, R, L)$ where

1. S is a finite set of states.
2. $I \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
4. $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

Temporal logic is used to specify properties of Kripke structures. $CTL^{\star}$, a powerful logic, describes properties of computation trees. A tree is formed by designating a state in a Kripke structure as the initial state and then unwinding the structure into an infinite tree with the designated state at the root. In $CTL^{\star}$, formulas are composed of path quantifiers and temporal operators. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers A (for all computation paths) and E (for some computation path). The temporal operators, X (next time), F (in the future), G (always), U (until), and R (release) describe properties of a path through the tree.

There are two types of formulas in $CTL^{\star}$: state formulas which are true in a specific state and path formulas which are true along a specific path. Let AP be the set of atomic propositions, the syntax of $CTL^{\star}$ is given by the following rules:

1. If $p \in AP$, then p is a state formula.
2. If f and g are state formulas, then $\neg f, f \wedge g$, and $f \vee g$ are state formulas.
3. If f is a path formula, then Ef and Af are state formulas.
4. If f is a state formula,then f is also a path formula.
5. If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, Xf, Ff, fUg, and fRg are path formulas.

Let M be a Kripke structure over AP. A path in M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \cdots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$. We use π^i to denote the suffix of π starting at s_i.

The restriction of $CTL^\star$ to universal path quantifiers A is called $ACTL^\star$.

Simulation equivalence restricts the logic and relaxes the requirement that the structures should satisfy exactly the same formulas, resulting in a great reduction.

Definition 2 (Simulation relation) Given two structures M and M' with $AP' \subseteq AP$, a relation $H \subseteq S \times S'$ is a *simulation relation* between M and M' if and only if for all s and s', if $H(s, s')$ then the following conditions hold:

1. $L(s) \cap AP' = L'(s')$.
2. For every state s_1 such that $R(s, s_1)$, there is a state s'_1 with the property that $R'(s', s'_1)$ and $H(s_1, s'_1)$.

If there exists a simulation relation H such that for every initial state s_0 in M there is an initial state s'_0 in M' for which $H(s_0, s'_0)$, we say that M' *simulates* M (denoted by $M \preceq M'$).

4 Modeling parameterized systems

States of each process in a parameterized system are considered as interpretations over a finite variable set, V. For each V, a subset V^e is called an external variable set that is used by the process to communicate with the environment consisting of other processes. The set $V^i = V - V^e$ is an internal variable set. Obviously, the environment may update only external variables, whereas the process may update all the variables. Such processes are modeled by Kripke structures which describe a class of finite state systems with first-order logic propositions. A complex parameterized system is modeled as a composition of such smaller processes when the following conditions are met.

Definition 3 (Compatible structure) Two Kripke structures $M_1 = (AP_1, S_1, I_1, R_1, L_1)$ and $M_2 = (AP_2, S_2, I_2, R_2, L_2)$ are involved, in which V_1 and V_2 are their respective state variable sets. If $V_1^i \cap V_2^i = \varnothing$ and $V_1^e = V_2^e$ are true, then M_1 and M_2 are compatible structures. The former condition indicates that internal variables are owned only by one process and the latter requires external variables shared by both processes.

Definition 4 (Compatible state) Let $M_1 = (AP_1, S_1, I_1, R_1, L_1)$ and $M_2 = (AP_2, S_2, I_2, R_2, L_2)$ be two compatible structures. If $L_1(s_1) \cap AP_2 = L_2(s_2) \cap AP_1$ is true, then $s_1 \in S_1$ and $s_2 \in S_2$ are compatible. Compatible states agree on the external variables as well as the common atomic propositions.

Processes communicate with each other in the synchronous or asynchronous mode. In the synchronous execution mode, all processes execute the transitions at the same time, whereas in the asynchronous execution mode, the process state transitions are independent of each other: the system evolves by interleaving the evolution of its processes. At each execution cycle, only one process is chosen to perform a transition. However, parameterized systems, in which different processes may change their states at the same time, are very common in reality. There is no order between these transitions, thus preserving the true meanings of concurrency. We call such a communication mode as asynchronous composition with true concurrency semantics. From the viewpoint of computer science, it is more interesting to investigate asynchronous products of Kripke structures with true concurrency semantics. We propose a formal model with true concurrency semantics for parameterized systems, which is more suitable for describing concurrent systems in the usual fashion.

Definition 5 (Asynchronous composition with true concurrency semantics) Let $M_k = (AP_k, S_k, I_k, R_k, L_k)$ be the kth $(1 \leq k \leq n)$ Kripke structure among compatible structures. Their asynchronous composition with true concurrency semantics,

$$M = \prod_{k=1}^{n}{}_{a} M_k = (AP, S, I, R, L)$$

is defined to be:

1. $AP = \bigcup_{k=1}^{n} AP_k$.
2. $S = \{< s_1, s_2, \ldots, s_n > | s_k \in S_k\ (1 \leq k \leq n)\ \textit{are compatible states}\} \subseteq \prod_{k=1}^{n} S_k$.
3. $I = \{< s_1, s_2, \ldots, s_n > | \bigwedge_{k=1}^{n} s_k \in I_k\} \subseteq S$.
4. $R = \{(< s_{1,i}, s_{2,i}, \ldots, s_{n,i} >, < s_{1,i+1}, s_{2,i+1}, \ldots, s_{n,i+1} >) | \exists j, 1 \leq j \leq n, (s_{j,i}, s_{j,i+1}) \in R_j\}$.
5. $L(< s_1, s_2, \ldots, s_n >) = \bigcup_{k=1}^{n} L_k(s_k)$.

Theorem 1 *The asynchronous composition operator with true concurrency semantics, $\prod_a$, is commutative and associative.*

Proof By Definition 5, the set of atomic propositions of the composition is a union of component atomic propositions; so is the set of labels. States of the composition are vectors of component states that are compatible, and they are elements of the Cartesian product of component states. Each transition of the composition involves at least a transition of n components. Because the union and product of sets are commutative and associative, the asynchronous composition operator with true concurrency semantics is also commutative and associative. $\square$

5 Two-dimensional abstraction

Now we use a two-dimensional graph shown in Fig. 1 to describe the state space of parameterized systems, where the x axis denotes system parameter n, and the y axis denotes the state space of each process m. To simplify the presentation, it is supposed that all processes are identical. Since the full cross-product of the process states needs to be considered in the global system at each step, the result of the asynchronous composition with true concurrency semantics is very large, in the worst case m^n.

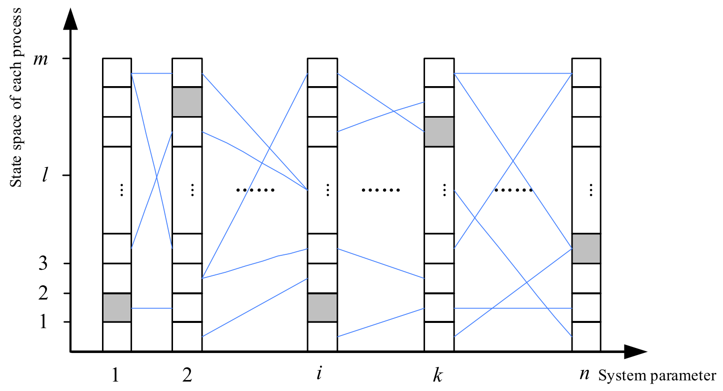


Fig. 1 State space of parameterized systems

Too many reachable states impede the automatic verification in many practical cases. Two-dimensional abstraction technique proposed in this paper is specifically tailored for parameterized systems with true concurrency semantics and helps avoiding the problem of state explosion.

Definition 6 (Two-dimensional abstraction) For asynchronous concurrent parameterized systems with true concurrency semantics, *two-dimensional abstraction* is a process constructing an abstract model by first reducing the state space of each process independently along the y axis in order to reduce m and then hiding the system parameter n along the x axis based on the design principles of parameterized systems. The former step is called *y-abstraction*, and the latter *x-abstraction*. The corresponding reduced results are called the *y-abstract* model and *TDA* model, respectively.

The selection of an equivalence relation between a TDA model and a concrete system is of prime importance for the successful application of TDA in practice. Simulation relationship [35] will result in a greater reduction of the number of states by restricting logic and relaxing the requirement that two structures should satisfy exactly the same set of formulas. Given two Kripke structures $M_1 = (AP_1, S_1, I_1, R_1, L_1)$ and $M_2 = (AP_2, S_2, I_2, R_2, L_2)$ with $AP_2 \subseteq AP_1$, if there exists a simulation relation H such that for every initial state s_{10} ($s_{10} \in I_1$) in M_1 there is an initial state s_{20} ($s_{20} \in I_2$) in M_2 for which $H(s_{10}, s_{20})$, we say that M_2 simulates M_1 and denote it by $M_1 \preccurlyeq M_2$. Intuitively, for every transition in M_1, there is a corresponding transition in M_2.

In the following sections, $PS^c(n)$ refers to the concrete model of asynchronous concurrent parameterized systems with true concurrency semantics consisting of n concrete processes. $PS^y(n)$ is the y-abstract model of $PS^c(n)$ and $PS^t(n)$ is its TDA model.

5.1 y-Abstraction

The y-abstraction deals with each concrete process independently in order to abstract away the information irrespective of system properties. Any property-preserving abstraction method is available. We construct a finite predicate set $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_r\}$ from properties and system description, and build the y-abstract model through the method of basic predicate abstraction.

The predicate set Φ defines an equivalence relationship on S_k^c, the set of states of $M_k^c = (AP_k^c, S_k^c, I_k^c, R_k^c, L_k^c)$ $(1 \le k \le n)$, and each equivalence class is denoted by an abstract state. The concrete state is labeled with a predicate formula which is satisfied in that state. In other words, labeling function L_k^c maps a concrete state into a predicate set. The set of states of the y-abstract model M_k^y, S_k^y is a set of normal boolean expressions on $b_1, b_2, \ldots, b_r$ $(b_j (1 \le j \le r)$ corresponding to predicate φ_j. A y-abstract state is a truth assignment to r boolean variables. Labeling function L_k^y maps a y-abstract state into a boolean expression. The abstract operator H_k^{cy} determines the relationship between concrete states and abstract states. The method of building the transition relation R_k^y of the y-abstract model M_k^y from the concrete transition relation R_k^c is the same as that introduced by Graf and Saidi [11]. From the above definitions, we can conclude that $H_k^{cy} \subseteq S_k^c \times S_k^y$ is a simulation relation between M_k^c and M_k^y, so the following theorem holds.

Theorem 2 $M_k^c \preccurlyeq M_k^y$ $(1 \le k \le n)$.

Proof The proof is given in [11]. □

In the following, we will demonstrate how the y-abstraction affects the parameterized concurrent systems.

Definition 7 (Visible transitions set and invisible transitions set) Given a Kripke structure $M = (AP, S, I, R, L)$, we assume that AP_f is the set of atomic propositions involved in the temporal formula f. The *set of visible transitions* of M w.r.t. AP_f includes transitions affecting the truth of atomic propositions in AP_f, which is denoted by $VTS(M, AP_f) = \{(s,t) | (s,t) \in R \wedge (L(s) \cap AP_f \ne L(t) \cap AP_f)\}$. The set of $IVTS(M, AP_f) = R - VTS(M, AP_f)$ is called the *set of invisible transitions* of M w.r.t. AP_f.

It is obvious that $VTS(M, AP_f)$ and $IVTS(M, AP_f)$ relate to the system property. Both of them satisfy $VTS(M, AP_f) \cap IVTS(M, AP_f) = \varnothing$ and $VTS(M, AP_f) \cup IVTS(M, AP_f) = R$.

For each transition $R_k^c(s_k^c, t_k^c)$ of the kth process in a concrete model $PS^c(n)$, if $R_k^c(s_k^c, t_k^c) \in IVTS(M_k^c, AP_f)$, the corresponding y-abstract transition $R_k^y(s_k^y, t_k^y)$ is a loop in the state graph and M_k^y does not change the current state. If $R_k^c(s_k^c, t_k^c) \in VTS(M_k^c, AP_f)$, $R_k^y(s_k^y, t_k^y)$ connects two different y-abstract states in M_k^y, that is to say, M_k^y performs a transition. Hence, all transitions in M_k^c are maintained. Figure 2 illustrates two kinds of concrete transitions and their y-abstract transitions. Therefore, the y-abstract model $PS^y(n)$ is an asynchronous composition of $M_1^y, M_2^y, \ldots, M_n^y$.

Theorem 3 *The asynchronous composition with true concurrency semantics operator* $\prod_a$ *is monotonic w.r.t.* $\preccurlyeq$, *that is,* $M_k^c \preccurlyeq M_k^y$ $(1 \le k \le n) \Rightarrow PS^c(n) \preccurlyeq PS^y(n)$.

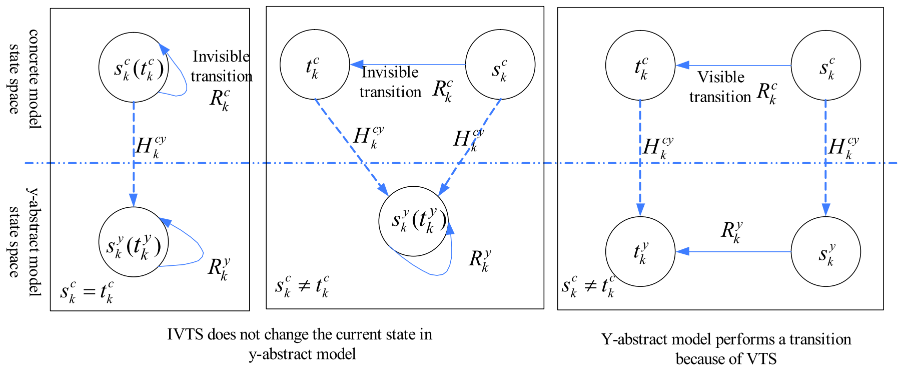

concrete model state space
y-abstract model state space
$s_k^c(t_k^c)$
Invisible transition R_k^c
H_k^{cy}
$s_k^y(t_k^y)$
R_k^y
$s_k^c = t_k^c$
t_k^c
Invisible transition R_k^c
s_k^c
H_k^{cy}
H_k^{cy}
$s_k^y(t_k^y)$
R_k^y
$s_k^c \neq t_k^c$
t_k^c
Visible transition R_k^c
s_k^c
H_k^{cy}
H_k^{cy}
R_k^y
t_k^y
s_k^y
$s_k^c \neq t_k^c$
IVTS does not change the current state in y-abstract model
Y-abstract model performs a transition because of VTS

Fig. 2 How y-abstraction affects transitions in asynchronous concurrent parameterized systems

Proof Let $PS^c(n) = (AP^c, S^c, I^c, R^c, L^c) = \prod_a^n {}_{k=1} M_k^c$ be an asynchronous composition with true concurrency semantics, where $M_k^c = (AP_k^c, S_k^c, I_k^c, R_k^c, L_k^c)$. Its y-abstract model is denoted by $PS^y(n) = (AP^y, S^y, I^y, R^y, L^y) = \prod_a^n {}_{k=1} M_k^y$, where $M_k^y = (AP_k^y, S_k^y, I_k^y, R_k^y, L_k^y)$.

First of all, from Theorem 2, we have

$$M_k^c \preccurlyeq M_k^y. \tag{1}$$

Therefore,

$$AP_k^y \subseteq AP_k^c. \tag{2}$$

By Definition 5, it is easy to see that

$$AP^y = \bigcup_{k=1}^{n} AP_k^y \subseteq \bigcup_{k=1}^{n} AP_k^c = AP^c. \tag{3}$$

Note that the abstract function H_k^{cy}, described in Sect. 5.1, is a simulation relation between M_k^c and M_k^y, hence, for every s^y in $PS^y(n)$, the following identity holds:

$$\begin{aligned} s^y &= \langle s_{1a}^y, s_{2b}^y, \ldots, s_{kl}^y, \ldots, s_{ng}^y \rangle \\ &= \langle H_1^{cy}(s_{1a}^c), H_2^{cy}(s_{2b}^c), \ldots, H_k^{cy}(s_{kl}^c), \ldots, H_n^{cy}(s_{ng}^c) \rangle. \end{aligned} \tag{4}$$

That is to say, a y-abstract state is obtained by applying H_k^{cy} $(1 \le k \le n)$ to the kth element in concrete state s^c.

Now we will show that $H^{cy} \subseteq S^c \times S^y$ is a simulation relation between $PS^c(n)$ and $PS^y(n)$. For every $s^c = \langle s_{1a}^c, s_{2b}^c, \ldots, s_{kl}^c, \ldots, s_{ng}^c \rangle \in S^c$, suppose that $s^y = \langle s_{1a}^y, s_{2b}^y, \ldots, s_{kl}^y, \ldots, s_{ng}^y \rangle \in S^y$ is its y-abstract state, namely, $H^{cy}(s^c) = s^y$, then, by Definition 2, both of the following conditions must hold:

1. $L^c(s^c) \cap AP^y = L^y(s^y)$.
2. $\forall t^c\ t^c \in S^c \wedge R^c(s^c, t^c) \Rightarrow \exists t^y\ t^y \in S^y \wedge R^y(s^y, t^y) \wedge H^{cy}(t^c, t^y)$.

Proof of condition (1): $L^c(s^c) \cap AP^y = L^y(s^y)$.

By Definition 5, observe that

$$\begin{aligned} L^c\big(s^c\big) \cap AP^y &= L^c\big(\big\langle s^c_{1a}, s^c_{2b}, \ldots, s^c_{kl}, \ldots, s^c_{ng}\big\rangle\big) \cap AP^y \\ &= \bigcup_{k=1}^{n} L^c_k\big(s^c_{kl}\big) \cap AP^y \\ &= \bigcup_{k=1}^{n} \big(L^c_k\big(s^c_{kl}\big) \cap AP^y\big). \end{aligned} \tag{5}$$

Because

$$AP^y = \bigcup_{k=1}^{n} AP^y_k, \tag{6}$$

if we replace AP^y in (5) with the right-hand side of (6), we obtain

$$L^c\big(s^c\big) \cap AP^y = \bigcup_{k=1}^{n} \left(L^c_k\big(s^c_{kl}\big) \cap \bigcup_{k=1}^{n} AP^y_k \right). \tag{7}$$

Note that $L^c_k(s^c_{kl})$ is a set of atomic propositions true in s^c_{kl}, so it is only relative to AP^c_k and independent of AP^c_j $(1 \le j \le n,\ j \ne k)$. Furthermore, $AP^y_k \subseteq AP^c_k$. Therefore,

$$\begin{aligned} L^c_k\big(s^c_{kl}\big) \cap \bigcup_{k=1}^{n} AP^y_k &= L^c_k\big(s^c_{kl}\big) \cap \big(AP^y_1 \cup \cdots \cup AP^y_k \cup \cdots \cup AP^y_n\big) \\ &= L^c_k\big(s^c_{kl}\big) \cap AP^y_k \\ &= L^y_k\big(s^y_{kl}\big). \end{aligned} \tag{8}$$

Substitute this item into (7) to obtain

$$L^c\big(s^c\big) \cap AP^y = \bigcup_{k=1}^{n} L^y_k\big(s^y_{kl}\big) = L^y\big(s^y\big). \tag{9}$$

Hence, condition (1) is true.

Proof of condition (2): $\forall t^c\ t^c \in S^c \wedge R^c(s^c, t^c) \Rightarrow \exists t^y\ t^y \in S^y \wedge R^y(s^y, t^y) \wedge H^{cy}(t^c, t^y)$.

For each $t^c = \langle t^c_{1a'}, t^c_{2b'}, \ldots, t^c_{kl'}, \ldots, t^c_{ng'} \rangle \in S^c$, $R^c(s^c, t^c)$ implies that there is at least one component in a concrete model that makes a transition. Suppose that the former k $(1 \le k \le n)$ components make transitions, while the latter $n-k$ components do not. There are several cases to be considered.

Case 1: $t^c \ne s^c \wedge R^c_k(s^c_{kl}, t^c_{kl'}) \in IVTS(M^c_k, AP_f)$, as represented in the middle of Fig. 2.

Because

$$M_k^c \preceq M_k^y, \tag{10}$$

one gets

$$R_k^c(s_{kl}^c, t_{kl'}^c) \Rightarrow \exists t_{ke'}^y \in S_k^y \; R_k^y(s_{ke}^y, t_{ke'}^y) \wedge H_k^{cy}(t_{kl'}^c, t_{ke'}^y). \tag{11}$$

Now we construct t^y by Definition 5 as follows:

$$\begin{aligned} t^y &= \langle t_{1a'}^y, \ldots, t_{kl'}^y, t_{(k+1)r'}^y, \ldots, t_{ng'}^y \rangle \\ &\triangleq \langle t_{1a'}^y, \ldots, t_{kl'}^y, s_{(k+1)r}^y, \ldots, s_{ng}^y \rangle \\ &= \langle H_1^{cy}(t_{1a'}^c), \ldots, H_k^{cy}(t_{kl'}^c), s_{(k+1)r}^y, \ldots, s_{ng}^y \rangle \\ &= \langle H_1^{cy}(t_{1a'}^c), \ldots, H_k^{cy}(t_{kl'}^c), H_{k+1}^{cy}(s_{(k+1)r}^c), \ldots, H_n^{cy}(s_{ng}^c) \rangle. \end{aligned} \tag{12}$$

As the latter $n-k$ components in the concrete model do not make transitions, we obtain

$$s_{(k+1)r}^c = t_{(k+1)r'}^c, \ldots, s_{ng}^c = t_{ng'}^c. \tag{13}$$

Substitute them into (12) to obtain

$$t^y = \langle H_1^{cy}(t_{1a'}^c), \ldots, H_k^{cy}(t_{kl'}^c), H_{k+1}^{cy}(t_{(k+1)r'}^c), \ldots, H_n^{cy}(t_{ng'}^c) \rangle. \tag{14}$$

This expression indicates that applying H_k^{cy} to the kth element of t^c will yield its y-abstract state, thus, $(t^c, t^y) \in H^{cy}$.

From (11), there is at least one element in s^y and t^y that satisfies $R_k^y(s_{ke}^y, t_{ke'}^y)$, so $(s^y, t^y) \in R^y$.

The other two cases, $t^c \neq s^c \wedge R_k^c(s_{kl}^c, t_{kl'}^c) \in VTS(M_k^c, AP_f)$ and $t^c = s^c$, can be discussed in a similar way.

To this point, both conditions (1) and (2) are true. We conclude that $H^{cy} \subseteq S^c \times S^y$ is a simulation between $PS^c(n)$ and $PS^y(n)$. By Definition 2, for every initial state $s_0^c \in I^c$ in $PS^c(n)$ there is an initial state $s_0^y \in I^y$ in $PS^y(n)$ such that $H^{cy}(s_0^c, s_0^y)$, as a consequence, this theorem is proved. □

Theorem 3 implies that the y-abstract model is weakly-preserved w.r.t. *ACTL** formula. Applying this theorem to each kind of *ACTL** formula, we get the following conclusion.

Theorem 4 *For each ACTL* formula* $f(AP_f \subseteq AP^y)$, $PS^y(n) \models f \Rightarrow PS^c(n) \models f$.

Proof From Theorem 3, we obtain

$$PS^c(n) \preceq PS^y(n).$$

Hence, $PS^y(n) \models f \Rightarrow PS^c(n) \models f$ holds. It is proved in [34]. □

Intuitively, this theorem is true because formula in *ACTL** describes properties that are quantified over all possible behaviors of a system. Because every behavior of $PS^y(n)$ is a behavior of $PS^c(n)$, every formula of *ACTL** that is true in $PS^y(n)$ must also be true in $PS^c(n)$. Theorem 4 is very useful for large scale system verification since it provides a way of accelerating the verification by taking advantage of exhaustive search of a smaller state space.

5.2 x-Abstraction

During the construction of parameterized systems, the designers reason about its correctness by focusing on the execution of one process (called *hub*) and consider its interaction with other processes (called *rims*, all *rims* constitute the *hub*'s environment) [8]. The x-abstraction, following this idea, produces a much smaller state space.

As described in the earlier sections, $PS^y(n)$ is an asynchronous concurrent system with true concurrency semantics. Without loss of generality, assume that $PS^y(n)$ contains $n-1$ $(n>1)$ *rims* (numbered from 1 to $n-1$) and one *hub* (numbered n). We get the following identity by expanding L^y, the labeling function of $PS^y(n)$:

$$\begin{aligned}&L^y\left(\langle s_1^y,\ldots,s_k^y,\ldots,s_{n-1}^y,s_h^y\rangle\right)\\&\quad=L_1^y\left(s_1^y\right)\cup\cdots\cup L_k^y\left(s_k^y\right)\cup\cdots\cup L_{n-1}^y\left(s_{n-1}^y\right)\cup L_h^y\left(s_h^y\right).\end{aligned}$$

It is straightforward to find that $L_k^y(s_k^y)$ $(1\le k\le n)$ on the right hand side of the identity is the set of all labels of *rims* (or *hubs*) and they are atomic propositions that process k satisfies in the current state. These atomic propositions reflect process properties. Consequently, the object of x-abstraction is the whole parameterized system whose properties relate to either one process or many processes.

Definition 8 (Process property) The first-order predicate $prop(k)$, $1\le k\le n$, indicating that the kth process has property *prop*, is called process property. We use $PROP(k)=\{prop(k)\}$ to denote all properties the kth process holds.

Given a process d, the *d-label* is an instance of $prop(k)$, meaning that process d meets the property *prop*. $PROP(d)=\{prop(d)\}$ is the set of all *d-labels*. For every s^y $(s^y\in S^y)$ and process d $(1\le d\le n)$, we have either $s^y\models prop(d)$ or $s^y\not\models prop(d)$. If $s^y\models prop(d)$ holds, the y-abstract state s^y has the label $prop(d)$.

The global state label of the y-abstract model can be simplified as follows, by Definition 8:

$$L^y=L(1)\cup\cdots\cup L(k)\cup\cdots\cup L(n)=\bigcup_{k=1}^{n}L(k)=\left\{l(d),s^y\models l(d),1\le d\le n\right\}. \quad (15)$$

It is interesting to note that the global label of the y-abstract state s^y is all the process properties it satisfied. Next we will introduce a new notation to describe the parameterized system.

Definition 9 The first-order predicate $snps(k)=prop(k)\wedge(\bigwedge_{j\ne k}prop(j))$ describes not only the kth process but also its environment (comprising the jth process). $snps(k)$ is a quite detailed picture of the global system, and all the snapshots are represented as $SNPS=\{snps(k)\}$.

A snapshot $\mathit{snps}(k)$ gives the necessary condition that an equivalent partition meets on $\mathit{PS}^y(n)$: if there exits a process d satisfying $s^y \models \mathit{snps}(d)$, $\mathit{snps}(k)$ is one of the abstract states of s^y. All such y-abstract states which satisfy the above condition compose an equivalence class. If $\mathit{snps}(k)$ were of the form $\pm \mathit{prop}_1(k) \wedge \pm \mathit{prop}_2(k) \wedge \cdots \wedge \pm \mathit{prop}_r(k)$, $r > 1$, where $\mathit{prop}_1(k), \ldots, \mathit{prop}_r(k)$ are r process properties and $\pm \mathit{prop}_i(k)$ $(1 \le i \le r)$ indicates that $\mathit{prop}_i(k)$ appears positive or negative, $\mathit{snps}(k)$ can be expressed by a tuple $\langle b_1, b_2, \ldots, b_r \rangle$, where $b_i = 1 \Leftrightarrow \mathit{snps}(k) \Rightarrow \mathit{prop}_i(k)$. That is, the value of each bit b_i reflects the polarity of the corresponding predicate $\mathit{prop}_i(k)$ in $\mathit{snps}(k)$. Labeling the y-abstract states with atomic formulas will result in a much smaller state space.

In order to construct a TDA model, *PROP* and *SNPS* must meet two conditions: coverage and congruence. Coverage means that every y-abstract state is reflected by some snapshots, and congruence implies that $\mathit{snps}(k)$ contains enough information about a process to conclude a label holds true for this process or not. That is to say, for each $\mathit{snps}(k) \in \mathit{SNPS}$ and each $\mathit{prop}(k) \in \mathit{PROP}$ it holds that $\mathit{snps}(k) \to \mathit{prop}(k)$ or $\mathit{snps}(k) \to \neg \mathit{prop}(k)$.

Suppose that *PROP* and *SNPS* of $\mathit{PS}^y(n)$ satisfy the above conditions, the TDA model is a Kripke structure $\mathit{PS}^t = \langle \mathit{AP}^t, S^t, I^t, R^t, L^t \rangle$:

1. AP^t is the set of atomic propositions involved in the process property $\mathit{prop}(k)$, and $\mathit{AP}^t = \mathit{AP}^y$ according to Definition 8;
2. $S^t = \mathit{SNPS}$ is the set of abstract states: the abstract operator $\alpha_n(s^y) = \{\mathit{snps}(k) \in \mathit{SNPS} | s^y \models \mathit{snps}(n)\}$ maps all the y-abstract states s^y, where *hub* meets the condition of $\mathit{snps}(k)$, into the TDA abstraction state $\mathit{snps}(k)$;
3. I^t is the set of initial abstract states: $\mathit{snps}(k) \in I^t$ if there exists a parameterized system $\mathit{PS}^y(n)$ and a y-abstract state $s^y \in I^y$ such that $\mathit{snps}(k) \in \alpha_n(s^y)$;
4. L^t is the labeling function: for each $\mathit{snps}(k) \in S^t$,$L^t(\mathit{snps}(k)) = \{\mathit{prop}(k) : \mathit{snps}(k) \Rightarrow \mathit{prop}(n)\}$;
5. R^t is the set of abstract transitions: for each $\mathit{snps}_1(k) \in S^t$, $\mathit{snps}_2(k) \in S^t$, if there exist a parameterized system $\mathit{PS}^y(n)$ and two y-abstract states $s^y \in S^y, t^y \in S^y$ which meet the condition of $\mathit{snps}_1(k) \in \alpha_n(s^y) \wedge \mathit{snps}_2(k) \in \alpha_n(t^y) \wedge (s^y, t^y) \in R^y$, then $(\mathit{snps}_1(k), \mathit{snps}_2(k)) \in R^t$.

The TDA abstract state is labeled with $\mathit{prop}(k)$ which process k satisfies, and now k becomes finite after y-abstraction, therefore, S^t is finite, too. From the theoretical perspective, TDA will reduce the space by $(|S| - |S^t|)/|S|$ where S is the set of asynchronous composition states defined in Definition 5. At this time, our goal of reducing the state space of parametric verification has been achieved.

Theorem 5 *For a single-indexed ACTL* specification* $\forall x\, \varphi(x)$ *where the atomic formulas involved in* $\varphi(x)$ *are labels in* L^t, *the following holds*: $\mathit{PS}^t \models \varphi(x) \Rightarrow \forall n\, \mathit{PS}^y(n) \models \forall x \varphi(x)$.

Proof The proof is given in [36]. □

The correctness of TDA means that TDA model is weakly-preserved for single indexed *ACTL** specifications, which is guaranteed by Theorems 3, 4, and 5. In addition, Theorem 5 implies that TDA is sound, namely, any single-indexed *ACTL** specification which holds in a TDA model also holds in a concrete model with arbitrary number of processes. The completeness and soundness of our approach provide a solid theoretical foundation for optimizing the state space of parameterized systems.

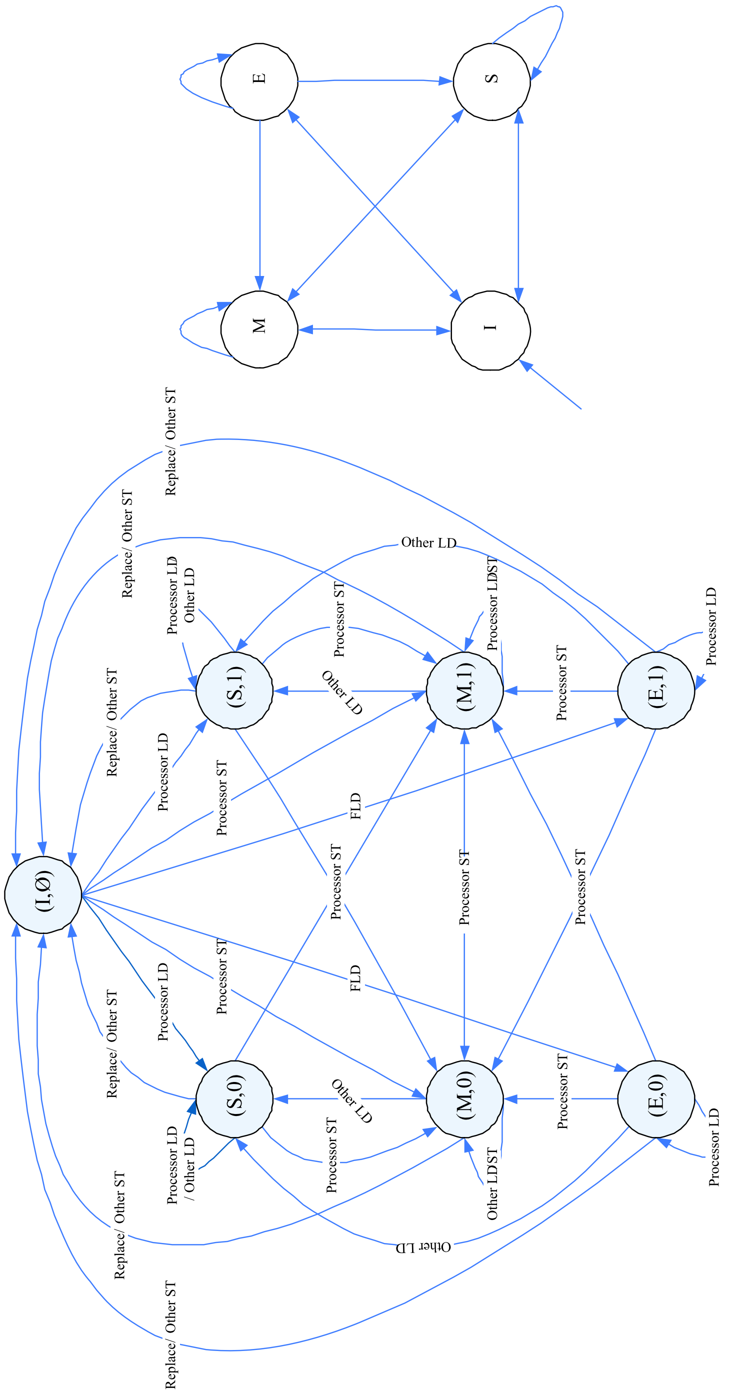
6 An example

We show how the TDA runs on parameterized MESI protocol. The MESI protocol is a four-state write-invalidate cache coherence protocol in which every memory block can be in one of the following states: *Modified*, *Exclusive*, *Shared*, and *Invalid* [37]. *Invalid* means that a memory block is not present in the cache and to load it the processor would have to send a request (LD) to the main memory. *Modified* identifies cache lines that have been written by the corresponding processor (ST). The current version of the modified block resides in the cache and is not visible to the rest of the system at this time. The processor can perform LD, ST, and Eviction on this data. *Shared* is the only state which allows other valid copies of the same memory block to be stored in other caches. A processor can load from a *Shared* memory block or evict it without notifying other processors or the memory. *Exclusive* means that the processor is the one who owns the right to modify the block and the main memory is current with the contents of the cache. If one cache has an *Exclusive* or *Modified* state, all matching lines in other caches are marked *Invalid*.

Let $PS^c(3)$ be a distributed shared-memory multi-processor system with three processors which ensures the data consistency through a directory-based MESI protocol considering single memory block and single cache line. The directory itself is a data structure whose entries record, for every block of memory, the state (i.e., cache access permission, namely, *dirstate*) and the identities of the processors which have cached that block (*sharedset*). Each cache tag residing in a processor includes at least three fields: *memaddr*, *cachestate*, and *cachedata*. From the viewpoint of each cache controller, a particular memory block can be in one of the four states: *MODF*, *EXCL*, *SHRD*, or *INVD*. From the perspective of system-wide view, the state of a cache line is determined by the corresponding *dirstate* and *cachestate*. Regardless of *dirstate*, if the range of *cachedata* is contained in [0, 1], there are as many as 32 transitions in the state machine of a single processor for a single memory block, even though 7 states are valid (shown on the left hand side of Fig. 3). It is very difficult to draw the state machine graph if *cachedata* and *memaddr* are allowed to take on any values from its domain.

Now we want to validate $PS^c(3)$ which satisfies such a property that there exists a processor without a copy of a block of memory when it is shared by another processor. The first step is to simplify the MESI protocol for a single processor through y-abstraction by Definition 6. Because the above property only relates to the state of cache line and does not care its value, *cachedata* is redundant. The Kripke structure of the reduced MESI protocol by y-abstraction is shown on the right hand side of Fig. 3, where states are labeled with predicates satisfied in the current state, for example, 'M' means $cachestate = MODF$.

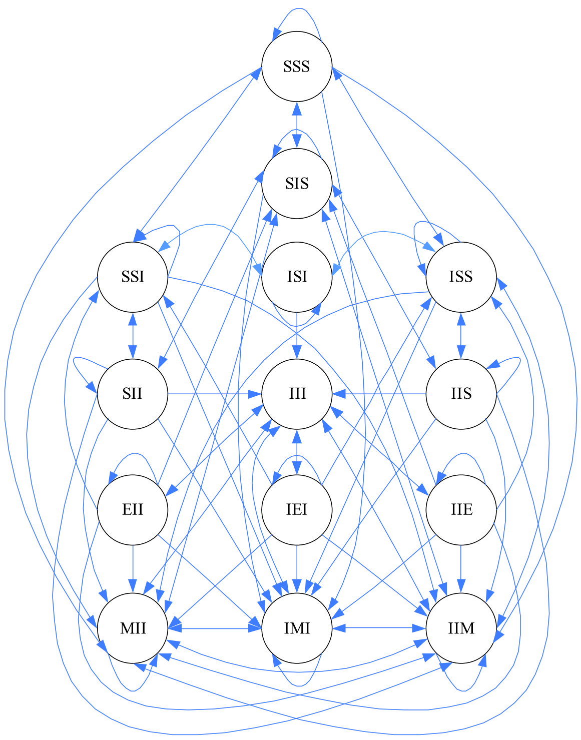
According to Definition 5, there are only 14 valid states out of a possible $4 * 4 * 4$ states in $PS^c(3)$ (shown in Fig. 4), each of them is labeled with a predicate-vector



FLD: First processor loading the block

Fig. 3 MESI state machine of a single processor for a single memory block, two values and its *y*-abstract model

Fig. 4 y-abstract model of 3 MESI-based processors



of length three, with the three bits representing the predicate the current memory block satisfies in processors 1, 2, and 3, respectively. For example, $\langle EII\rangle$ implies that processor 1 owns the right to modify the memory block and the memory data is not present in the caches of processors 2 and 3. To load the memory data, both of them must issue a request to the main memory. Other states are excluded due to compatibility constraints. Take $\langle MMM\rangle$ as an example. For the particular cache line in processors 1, 2 and 3, *cachestate* is an internal variable, whereas *dirstate* and *sharedset* are external variables. The labels for an M state in each processor are $\{dirstate = M, sharedset = P1, cachestate = M\}$, $\{dirstate = M, sharedset = P2, cachestate = M\}$, and $\{dirstate = M, sharedset = P3, cachestate = M\}$, respectively. The M states do not agree on the external variable *sharedset*, so they are not compatible.

In the second step, we use the following process property to represent that the block of memory is shared by the kth processor and there is another processor which has no copy of the block of memory:

$$\delta(k) = \big(cachestate[k] = S\big) \wedge \left(\bigvee_{j\neq k} cachestate[j] = I\right). \tag{16}$$

We define $prop_1(k)$ and $prop_2(k)$ by

$$prop_1(k) \triangleq \big(cachestate[k] = S\big), \tag{17}$$

Table 1 $PS^y(3)$ state space partition using $snps(2)$

Equivalence class	Label of equivalence class	Bit-vector of label
$\{ISI, SSI, ISS\}$	$prop_1(2) \wedge prop_2(2) = snps(2)$	$\langle 11 \rangle$
$\{III, IEI, IMI, EII, MII, IIE, IIS, IIM, SII\}$	$\neg prop_1(2) \wedge prop_2(2) = \neg snps(2)$	$\langle 01 \rangle$
$\{SSS\}$	$prop_1(2) \wedge \neg prop_2(2) = \neg snps(2)$	$\langle 10 \rangle$
$\{SIS\}$	$\neg prop_1(2) \wedge \neg prop_2(2) = \neg snps(2)$	$\langle 00 \rangle$

$$prop_2(k) \triangleq \exists j \big(j \neq k \wedge cachestate[j] = I \big). \tag{18}$$

Thus

$$\begin{aligned} snps(1) &= prop_1(1) \wedge prop_2(1), \\ snps(2) &= prop_1(2) \wedge prop_2(2), \\ snps(3) &= prop_1(3) \wedge prop_2(3). \end{aligned} \tag{19}$$

Table 1 demonstrates the result of the state space of $PS^y(3)$ partitioned by $snps(2)$. The first column lists the sets of equivalence class, while the second is the label of each equivalence class and its bit vector expression is shown in the last column. From the table we note that there are only 4 states in the TDA model, reducing the space by 71.4 % compared with that in the y-abstract model. The state of $\langle 11 \rangle$ in the resulting model means that processor 2 has a shared copy of the memory block and the memory data is not present in the caches of processor 1 and/or processor 3. Therefore, the TDA model is precise enough to prove the above system property, namely, TDA is correct.

Because the system parameter n is existentially-quantified, a group of parameterized systems with different system parameter can be modeled by the same TDA model. To prove the soundness, we applied our method to several other concrete systems. As it is expected, at least 3 concrete systems have the same TDA model as $PS^c(3)$ has. Figure 5 shows one such system.

7 Case studies

To validate our approach, we have implemented TDA and applied it to verify several classical cache coherence protocols as described in [38] and a hierarchical cache protocol in FT-1000 CPU.

7.1 Protocols and properties to be verified

Classical protocols and properties these protocols should have are introduced briefly here.

Synapse $N+1$

Synapse $N+1$ is a write-allocation protocol developed by Synapse for the $N+1$ computer. A cache can be in one of three possible states: *invalid* (the cache has no

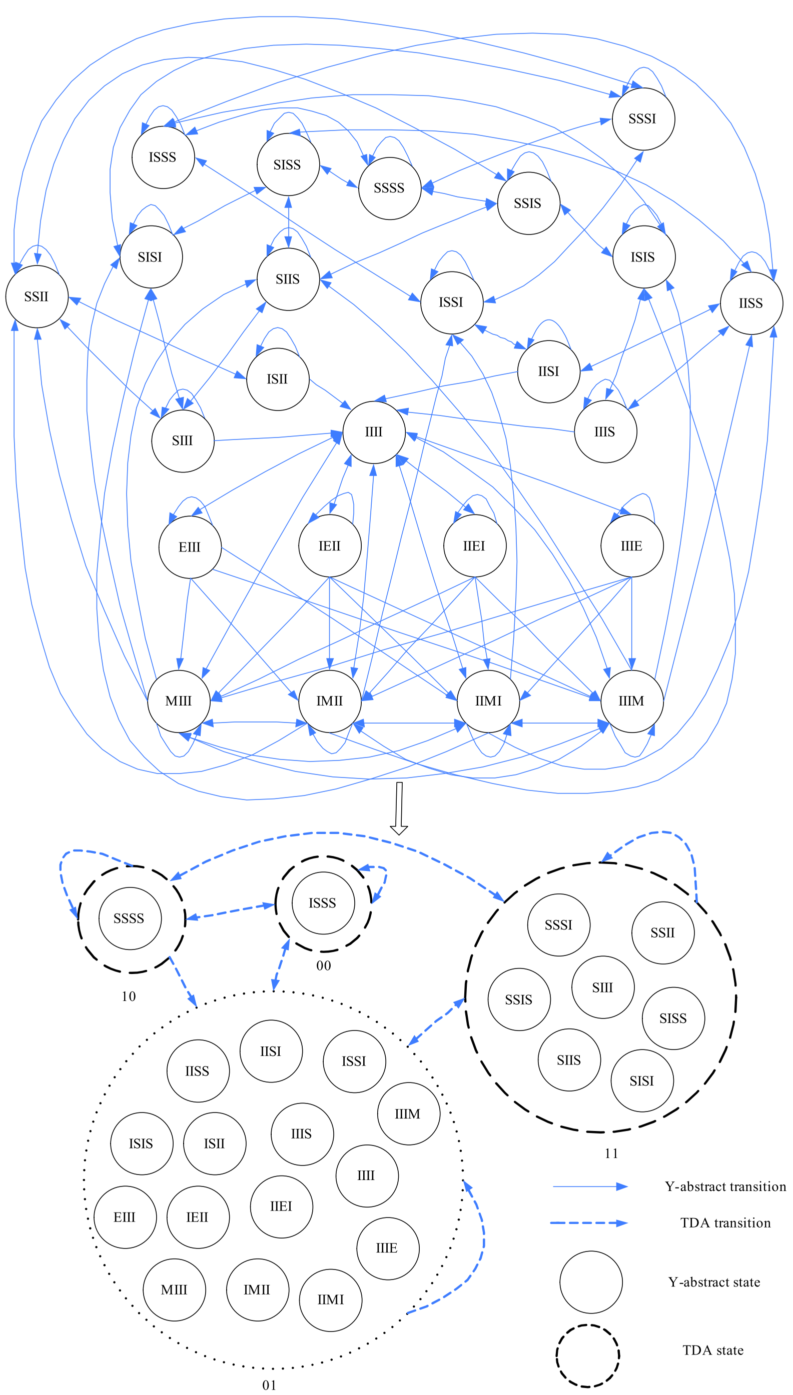


Fig. 5 Another concrete system with 4 MESI-based processors has the same TDA model as $PS^c(3)$ has

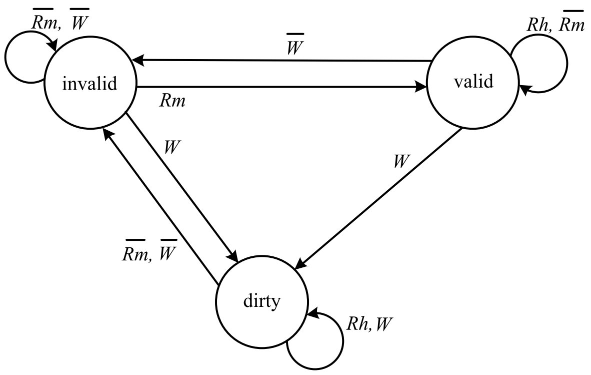

$\overline{Rm}$, $\overline{W}$
invalid
$\overline{W}$
valid
Rh, $\overline{Rm}$
Rm
W
W
$\overline{Rm}$, $\overline{W}$
dirty
Rh, W

Fig. 6 The Synapse $N+1$ protocol from the perspective of cache C_i

valid data), *valid* (the cache has a potentially shared copy of the data), and *dirty* (the cache has a modified copy of the data). *dirty* is an exclusive state, only one cache can have a dirty line. The state changes according to write and read commands issued by the corresponding processor (for example, R_m, W) or coming from the system bus (such as $\overline{R_m}$ and $\overline{W}$), as shown in Fig. 6, R_h is an internal action that denotes a read hit, R_m denotes a read miss, W denotes a write.

There are two possible sources of data inconsistency for Synapse:

UNS1: a *dirty* cache co-exists with one or more caches in state *valid*;

UNS2: more than one cache is in state *dirty*.

Illinois

The University of Illinois protocol is a snoopy cache, write-invalidate, write-in coherence policy. The special feature is that caches can have exclusive copies of data. Bus invalidation signals are sent only for writes to shared data. The memory copy is updated using a write-back policy (replacement). In addition to *invalid*, caches can be in one of the following states: *valid-exclusive* (the cache has an exclusive copy of the data that is consistent with the memory such that a modification of its content requires no bus invalidation signal), *shared* (the cache has a copy of the data consistent with the memory and other caches may have copies of the data), and *dirty* (the cache has a modified copy of the data, i.e., the data in main memory are obsolete and the content of the other caches is not valid). The transition is given in Fig. 7, and the behavior of one cache may be internal actions R_h (read hit), R_m (read miss), W_e (write in exclusive state), W_d (write in dirty state), *WI* (write and invalidate), and *Rep* (replacement with a new memory line). In this figure, P is defined as $Number(dirty) = 0 \wedge Number(shared) = 0 \wedge Number(valid\text{-}exclusive) = 0$, where $Number(q)$ denotes the number of caches in state q in the current global state.

The possible sources of data inconsistency are:

UNS1: a *dirty* cache co-exists with caches either in state *shared* or *valid-exclusive*;

UNS2: there is more than one *dirty* cache.

The other possible violations of the exclusivity of state *valid-exclusive* are:

UNS3: there is more than one *valid-exclusive* cache;

UNS4: a *shared* cache co-exists with a cache in state *valid-exclusive*.

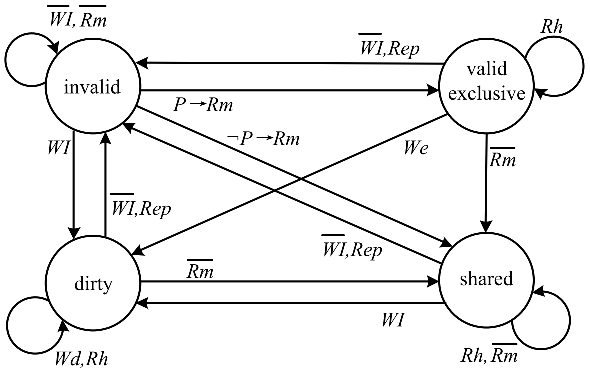


Fig. 7 The Illinois protocol from the perspective of cache C_i

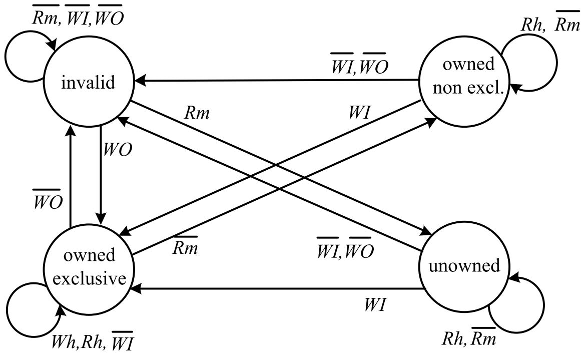


Fig. 8 The Berkeley protocol from the perspective of cache C_i

Berkeley

The Berkeley protocol is a variation of MESI with write-allocation and with a shared modified state, named *owned non-exclusively*. In this state, the main memory is not coherent with the possible multiple, cached copies of the owner data. The other three states are *invalid*, *unowned* (similar to the MESI *Shared* state), and *owned exclusively* (similar to the MESI *Modified* state). Figure 8 demonstrates how one cache changes its state according to different commands.

In the Berkeley protocol, we have the following sources of data inconsistency:

UNS1: an *owned exclusively* cache co-exists with one or more caches either in state *owned non-exclusively*, or *unowned*;

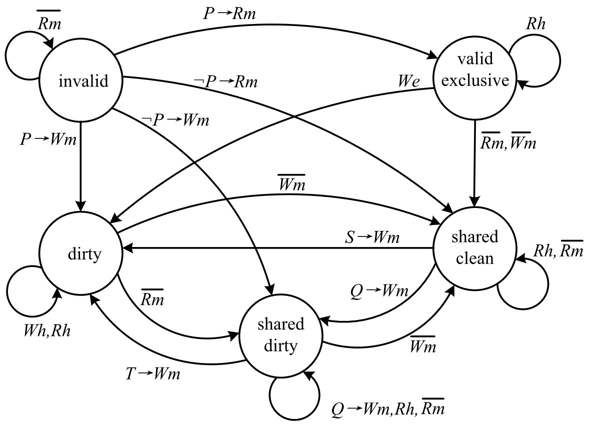
UNS2: there is more than one *owned exclusively* cache.

Dragon

Dragon is a write-allocation protocol that uses a signal to indicate snoop hits on the bus. The protocol has four states: *shared clean* (multiple clean copies may coexist), *shared dirty* (multiple dirty copies may coexist), *shared valid exclusive* (the cache has an exclusive clean copy), and *dirty* (the cache has an exclusive dirty copy). The possible transitions from the perspective of cache C_i are shown in Fig. 9, where P, Q, S, T are defined as follows:

$P \triangleq Number(exclusive) = 0 \wedge Number(dirty = 0) \wedge Number(shared\text{-}dirty) = 0 \wedge Number(shared\text{-}clean) = 0$,

Fig. 9 The Dragon protocol from the perspective of cache C_i



$Q \triangleq Number(shared\text{-}dirty) + Number(shared\text{-}clean) \geq 2,$
$S \triangleq Number(shared\text{-}dirty) = 0 \wedge Number(shared\text{-}clean) = 1,$
$T \triangleq Number(shared\text{-}dirty) = 1 \wedge Number(shared\text{-}clean) = 0.$

In the Dragon protocol, there are several possible sources of data inconsistency:

UNS1: a *dirty* cache co-exists with one or more caches either in state *shared dirty*, *shared clean* or *valid exclusive*;

UNS2: an *valid exclusive* cache co-exists with one or more caches either in state *shared clean*, or *shared dirty*;

UNS3: there is more than one *dirty* cache;

UNS4: there is more than one *valid exclusive* cache.

7.2 Experimental results

Figures 10 and 11 present some results of these experiments.

The asynchronous composition of n-processor system which ensures the data consistency through some protocol is a concrete system. Figure 10 shows the number of concrete states of each protocol against different system parameter according to Definition 5. Although in the worst case the number of states in asynchronous composition could be as large as $\prod_{k=1}^{n}|S_k|$, in practice it typically turns out to be much smaller. This is because some states, such as $\langle dirty, dirty\rangle$ in Illinois protocol and $\langle owned\text{-}exclusively, owned\text{-}exclusively\rangle$ in Berkeley protocol are prohibited. As it is seen from this figure, with the increase of processor number (especially greater than 13 for Berkeley and Dragon, 20 for Synapse $N+1$ and Illinois), the state number grows rapidly. Therefore, the largest asynchronous composition we can get only comprises 24 processors (Synapse $N+1$).

In Fig. 11, we plot the number of states in TDA model of each protocol. Because process properties used in TDA are made of predicates taken from properties to be verified, different properties for the same protocol have different TDA models. Two predicates, $cachestate(i) = dirty/shared$ and $Number(dirty/valid\text{-}exclusive)$, are enough to express these properties formally, resulting in 4, the maximum number of TDA abstract states. *AHG* denotes the number of reachable states in the abstract

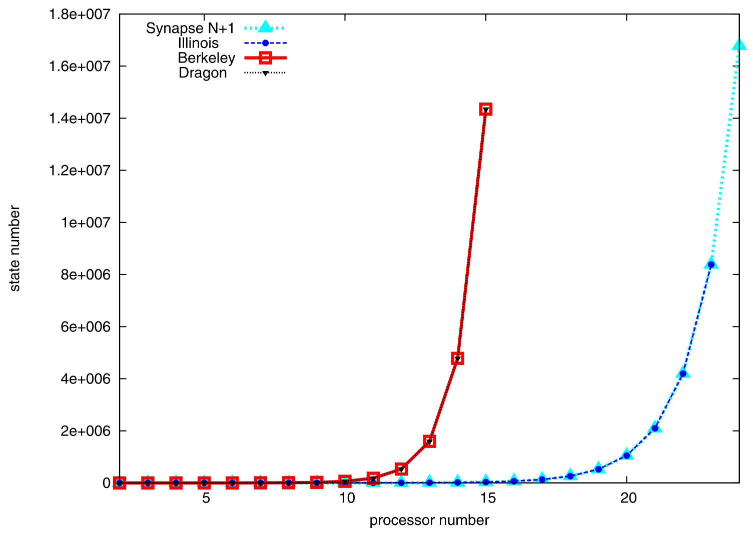


Fig. 10 Asynchronous composition state number with different processor number

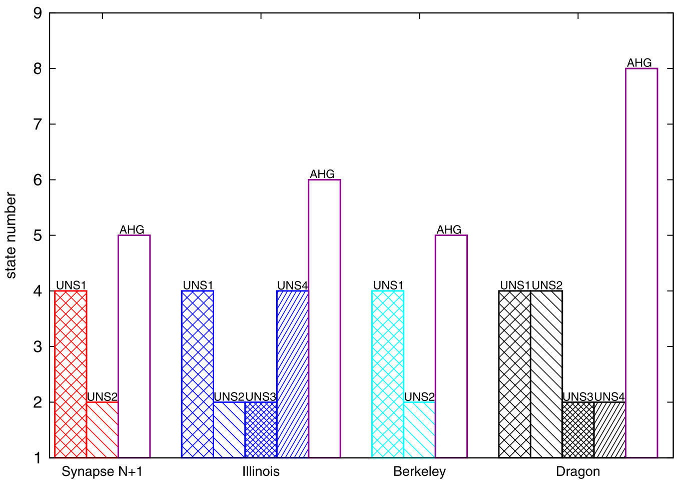


Fig. 11 TDA state number against properties, UNS1–UNS4 correspond to properties to be verified for each protocol, and AHG denotes abstract history graph

history graph described in [39] which are greater than those in TDA. It is also important to notice that the number of states in TDA model does not change along with the system parameter, which is consistent with the conclusion in Sect. 6. All experiments were conducted on a PC with a 3.3 GHz Intel Core processor, 8 Gb of available main memory, running Red Hat Linux (6.1) and GCC (4.4.5).

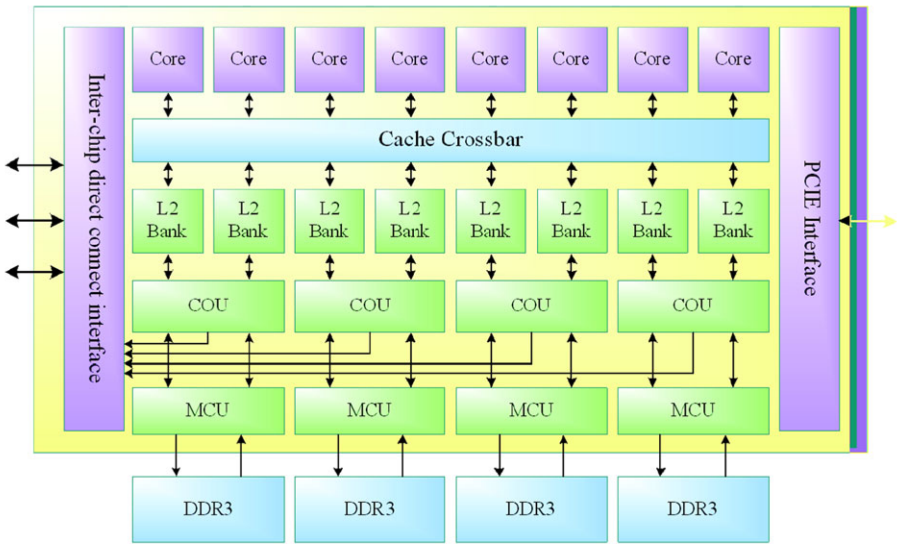


Fig. 12 Architecture of FT-1000 CPU

7.3 Application for FT-1000 CPU

FT-1000 CPU is a key component in TH-1A supercomputer system [40]. It adopts the parallel system on chip multi-core architecture. Eight multi-thread cores, each with a private cache hierarchy (L1 Cache), are integrated on the chip. The eight cores share a large capacity multi-bank L2 Cache, and communication between cores is achieved through Cache Crossbar. Cache Ordering Unit (COU) is responsible for cache coherence and memory ordering. L2 Cache can access the off-chip high speed DDR3 DRAM via memory controller units (MCU). The inter-chip direct connect interface supports cache coherence packet and large block data transfer packet, and can be used for connecting 2–4 processors directly to build large scale tightly-coupled shared-memory systems. This chip provides efficient I/O access by integrated PCIE 2.0 standard interface. Figure 12 illustrates the architecture of FT-1000 CPU.

In FT-1000 based SMP systems, a two-level hierarchical coherence protocol is designed to provide the coherent view of shared data items for programmers. The first level is the chip-level protocol used to keep multiple copies of the data among eight L1 caches consistent. The second level is the inter-chip protocol, used to maintain the L2 caches coherence among different chips. Both levels of this protocol are based on the standard three-state (*unowned*, *shared*, *exclusive*) invalidation-based directory-based cache coherence protocol with some extensions. This hierarchical protocol is more complicated, with more corner cases and bigger state space than non-hierarchical protocols, as we can see, it has eight instances of chip-level protocol and at most four instances of inter-chip protocol running concurrently. So it seems obvious that such hierarchical protocols cannot be checked by current model checkers, e.g., Murphi, NuSMV. During the development of FT-1000 CPU, we applied TDA to reduce the state space of chip-level protocol, and checked several safety properties using NuSMV. Then, FT-1000 CPU is regarded as a single-core processor and

Table 2 Experimental results of FT-1000 chip-level protocol

Asynchronous composition state number	TDA state number		Time (ms)	
	UNS1	UNS2	UNS1	UNS2
264	4	2	64	57

the verification of the inter-chip protocol is simplified. We claimed the correctness of the original protocol by verifying the second level protocol. Some chip-level experimental results are given in Table 2, where UNS1 and UNS2 are the same as those of Synapse $N+1$.

8 Conclusions

The verification of cache coherence in general is known to be NP-hard. In the age of exascale computing, scalability is emerging as one of the key components in parallel computing [41]. Scalable multi-core multi-processor architectures are inevitable. More and more complex processes and unbounded system parameter result in the state explosion during the verification of parameterized cache coherence protocols. A generic abstraction method for parameterized systems, two-dimensional abstraction (TDA), has been put forward in this paper. The novelty of our approach lies in that it analyzes in depth the intrinsic factors affecting the size of state space, and reduces the state space in two dimensions, thus a much smaller abstract model is produced. Compared with traditional approaches, our approach can effectively reduce the verification complexity and greatly scale the verification capabilities. We give complete soundness and completeness proofs for our method. We have demonstrated the benefits of our approach on several coherence protocols with realistic features.

Our future work is to integrate TDA with model-checking tools and check the advanced cache coherence protocol hierarchically organized for a next generation supercomputer. We also plan to investigate combining TDA with CMP method in the future.

Acknowledgements This work is inspired by the idea from M. Talupur's work on environment abstraction, and supported by the National Natural Science Foundation of China under Grant No. 61070036 and 61133007.

References

1. Grumberg O, Veith H (2008) 25 Years of model checking: history, achievements, perspectives. In: Lecture notes in computing science, vol 5000, VII. Springer, Berlin, p 231
2. Guerraoui R, Henzinger TA, Singh V (2008) Model checking transactional memories. Distrib Comput 22(3):129–145
3. Pong F, Dubios M (1997) Verification techniques for cache coherence protocols. ACM Comput Surv 29(1):82–126

4. Abts D, Lilja DJ, Scott S (2000) Toward complexity-effective verification: a case study of the cray SV2 cache coherence protocol. In: The 27th annual int'l symp on computer architecture (ISCA-2000), Vancouver, British Columbia, Canada
5. Abdulla PA, Delzanno G, Rezine A (2008) Monotonic abstraction in parameterized verification. Electron Notes Theor Comput Sci 223:3–14
6. Lahiri SK, Bryant RE (2007) Predicate abstraction with indexed predicates. ACM Trans Comput Logic 9(1). doi:10.1145/1297658.1297662
7. Pnueli A, Xu J, Zuck L (2002) Liveness with (0; 1; ∞) counter abstraction. In: Proc of the 14th international conference on computer aided verification (CAV). Lecture notes in computer science, vol 2404. Springer, Berlin, pp 107–122
8. Clarke E, Talupur M, Veith H (2006) Environment abstraction for parameterized verification. In: Proc of the 7th VMCAI. Lecture notes in computer science, vol 3855. Springer, Berlin, pp 126–141
9. Clarke E, Talupur M, Veith H (2008) Proving Ptolemy right: environment abstraction principle for parameterized verification. In: Proc of TACAS. Lecture notes in computer science, vol 4963. Springer, Berlin, pp 33–47
10. Pnueli A, Ruah S, Zuck L (2001) Automatic deductive verification with invisible invariants. In: Proc of the 7th TACAS, pp 82–97
11. Graf S, Saidi H (1997) Construction of abstract state graphs with PVS. In: Proc of the 9th int'l conf on computer aided verification. Springer, Berlin, pp 72–83
12. Sorin DJ, Plakal M, Condon AE, et al (2002) Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. IEEE Trans Parallel Distrib Syst 13(6):556–577
13. Dill DL, Drexler AJ, Hu AJ, Yang CH (1992) Protocol verification as a hardware design aid. In: IEEE intl conference on computer design, pp 522–525
14. Lamport L (2002) Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley, Reading
15. McMillan KL (2001) Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In: Proc on correct hardware design and verification methods (CHARME). Lecture notes in computer science, vol 2144. Springer, Berlin, pp 179–195
16. Chou C-T, Mannava PK, Park S (2004) A simple method for parameterized verification of cache coherence protocols. In: Proc on formal methods in computer-aided design (FMCAD). Lecture notes in computer science, vol 3312. Springer, Berlin, pp 382–398
17. Li Y (2007) Mechanized proofs for the parameter abstraction and guard strengthening principle in parameterized verification of cache coherence protocols. In: Proc of the ACM symposium on applied computing (SAC), pp 1534–1535
18. Chen X, Yang Y, Gopalakrishnan G, Chou C-T (2010) Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. Form Methods Syst Des 36(1):37–64
19. Krstic S (2005) Parameterized system verification with guard strengthening and parameter abstraction. In: Automated verification of infinite state systems (AVIS)
20. Talupur M, Krstic S, O'Leary J, Tuttle MR (2008) Parametric verification of industrial strength cache coherence protocols. In: Proc workshop on design of correct circuits (DCC)
21. Talupur M, Tuttle M (2008) Going with the flow: parameterized verification using message flows. In: Formal methods in computer aided design (FMCAD), pp 1–8
22. O'Leary J, Talupur M, Tuttle MR (2009) Protocol verification using flows: an industrial experience. In: Proc of the 9th international conference on formal methods in computer-aided design (FMCAD), pp 172–179
23. Pnueli A, Zuck LD (2003) Model-checking and abstraction to the aid of parameterized systems. In: Proc of the 4th international conference on verification, model checking, and abstract interpretation. Lecture notes in computer science, vol 2144. Springer, Berlin, p 4
24. Wang C, Hachtel GD, Somenzi F (2006) Abstraction refinement for large scale model checking. Springer, Berlin
25. Timm N, Wehrheim H (2010) On symmetries and spotlights—verifying parameterised systems. In: Lecture notes in computer science, vol 6447. Springer, Berlin, pp 534–548
26. Lahiri SK, Bryant RE, Cook B (2003) A symbolic approach to predicate abstraction. In: Proc of the international conference on computer-aided verification (CAV'03). Lecture notes in computer science, vol 2742. Springer, Berlin, pp 141–153
27. Konnov IV, Zakharov VA (2010) An invariant-based approach to the verification of asynchronous parameterized networks. J Symb Comput 45(11):1144–1162

28. Pandav S, Slind L, Gopalakrishnan G (2005) Counterexample guided invariant discovery for parameterized cache coherence verification. In: Proc of CHARME. Lecture notes in computer science, vol 3725. Springer, Berlin, pp 317–331
29. Delzanno G (2000) Automatic verification of parameterized of cache coherence protocols. In: Proc of the 12th international conference on computer aided verification (CAV), pp 53–68
30. Delzanno G, Bultan T (2001) Constraint-based verification of client–server protocols. In: Proc of the 7th international conference on principles and practice of constraint programming, pp 286–301
31. Delzanno G (2003) Constraint-based verification of parameterized cache coherence protocols. Form Methods Syst Des 23(3):257–301
32. Emerson EA, Kahlon V (2003) Exact and efficient verification of parameterized cache coherence protocols. In: Proc on correct hardware design and verification methods (CHARME'03). Lecture notes in computer science, vol 2860. Springer, Berlin, pp 247–262
33. Baukus K, Lakhnech Y, Stahl K (2002) Parameterized verification of a cache coherence protocol: safety and liveness. In: Proc VMCAI. Lecture notes in computer science, vol 2294. Springer, Berlin, pp 247–262
34. Clarke EM Jr, Grumberg O, Peled DA (1999) Model checking. The MIT Press, Cambridge, London
35. Girard A, Julius AA, Pappas GJ (2008) Approximate simulation relations for hybrid systems. Discret Event Dyn Syst 18(2):163–179
36. Talupur M (2006) Abstraction techniques for parameterized verification. Ph.D. Dissertation, Pittsburgh, Carnegie Mellon University
37. Culler D, Singh JP, Gupta A (1998) Parallel computer architecture: a hardware/software approach. Morgan Kaufmann, San Mateo
38. Delzanno G (2003) Constraint-based verification of parameterized cache coherence protocols. Form Methods Syst Des 23:257–301
39. Emerson EA, Kahlon V (2003) Rapid parameterized model checking of snoopy cache coherence protocols. In: Proc TACAS. Lecture notes in computer science, vol 2619. Springer, Berlin, pp 144–159
40. Yang X-J, Liao X-K, Lu K, et al (2011) The TianHe-1A supercomputer: its hardware and software. J Comput Sci Technol 26(3):344–351
41. Bosque JL, Roblas OD, Toharia P, Pastor L (2011) Evaluating scalability in heterogeneous systems. J Supercomput 58:367–375