D-Finder: A Tool for Compositional Deadlock Detection and Verification

Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis

Verimag Laboratory, Université Joseph Fourier Grenoble, CNRS

Abstract. D-Finder tool implements a compositional method for the verification of component-based systems described in BIP language encompassing multi-party interaction. For deadlock detection, D-Finder applies proof strategies to eliminate potential deadlocks by computing increasingly stronger invariants.

1 Methodology

Compositional verification techniques are used to cope with state explosion in concurrent systems. The idea is to aply divide-and-conquer approaches to infer global properties of complex systems from properties of their components. Separate verification of components limits state explosion. Nonetheless, components mutually interact in a system and their behavior and properties are inter-related. This is a major difficulty in designing compositional techniques [1,2,3,4,5,6,7,8]). As explained in [9], compositional rules are in general of the form

$$\frac{B_1 < \Phi_1 >,\ B_2 < \Phi_2 >,\ C(\Phi_1, \Phi_2, \Phi)}{B_1 \| B_2 < \Phi >} \tag{1}$$

That is, if two components with behaviors B_1, B_2 meet individually properties Φ_1, Φ_2 respectively, and $C(\Phi_1, \Phi_2, \Phi)$ is some condition taking into account the semantics of parallel composition operation and relating the individual properties with the global property, then the system $B_1 \| B_2$ resulting from the composition of B_1 and B_2 will satisfy a global property Φ.

In D-Finder, we implemented a novel approach for compositional verification of invariants based on the following rule:

$$\frac{\{B_i < \Phi_i >\}_i,\ \Psi \in II(\|_\gamma \{B_i\}_i, \{\Phi_i\}_i),\ (\bigwedge_i \Phi_i) \wedge \Psi \Rightarrow \Phi}{\|_\gamma \{B_i\}_i < \Phi >} \tag{2}$$

The rule allows to prove invariance of Φ for systems obtained by using a n-ary composition operation parameterized by a set of interactions γ. It uses global invariants which are the conjunction of individual invariants of components Φ_i and an interaction invariant Ψ. The latter expresses constraints on the global state space induced by interactions between components. It can be computed automatically from abstractions of the system to be verified. These are the composition of finite state abstractions B_i^α of the components B_i with respect to their

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 614–619, 2009.

invariants Φ_i. They can be represented as a Petri net whose transitions correspond to interactions between components. Interaction invariants correspond to traps [10] of the Petri net and are computed symbolically as solutions of a set of boolean equations.

Our method differs from assume-guarantee methods in that it avoids combinatorial explosion of the decomposition and is directly applicable to systems with multiparty (not only binary) interactions. Furthermore, it needs only guarantees for components. It replaces the search for adequate assumptions for each component by the use of interaction invariants. These can be computed automatically from given component invariants (guarantees). Interaction invariants correspond to a "*cooperation test*" in the terminology of [11] as they allow to eliminate product states which are not feasible by the semantics.

1.1 Checking Deadlock-Freedom and Invariance Properties

D-Finder provides a method for automated verification of component-based systems described in BIP (Behavior-Interaction-Priority) language [12]. In BIP, a system is the composition of a set of atomic components which are automata extended with data and functions written in C. To prove a global invariant Φ for a system $\gamma(B_1, \ldots, B_n)$, obtained by composing a set of atomic components $B_1, ..., B_n$ by using a set of interactions γ, we use the rule (2) above, where $B_i < \Phi_i >$ means that Φ_i is an invariant of component B_i and Ψ is an interaction invariant of $\gamma(B_1, \ldots, B_n)$ computed automatically from Φ_i and $\gamma(B_1, \ldots, B_n)$. A key issue in the application of this rule is finding component invariants Φ_i. If the components B_i are finite state, then we can take $\Phi = Reach(B_i)$, the set of reachable state of B_i, or any upper approximation of $Reach(B_i)$. If the components are infinite state, $Reach(B_i)$ is approximated using techniques presented in [13,14].

- **Checking Invariance Properties**. We give a sketch of a semi-algorithm allowing to prove invariance of Φ by iterative application of the rule (2). The semi-algorithm takes a system $\langle \gamma(B_1, \ldots, B_n), Init \rangle$ and a predicate Φ. It iteratively computes invariants of the form $\mathcal{X} = \Psi \wedge (\bigwedge_{i=1}^{n} \Phi_i)$ where Ψ is an interaction invariant and Φ_i an invariant of component B_i. If $\mathcal{X}$ is not strong enough for proving that Φ is an invariant ($\mathcal{X} \wedge \neg\Phi = false$) then either a new iteration with stronger Φ_i is started or we stop. In this case, we cannot conclude about invariance of Φ.
- **Checking Deadlock-Freedom**. Checking global deadlock-freedom of a system $\gamma(B_1, \ldots, B_n)$ is a particular case of proving invariants - proving invariance of the predicate $\neg DIS$, where DIS is the set of the states of $\gamma(B_1, \ldots, B_n)$ from which all interactions are disabled.

1.2 Generating Component Invariants and Interaction Invariants

D-Finder provides methods for computing component invariants, particulary useful for checking deadlock-freedom. It also provides a general method for computing interaction invariants for $\gamma(B_1, \ldots, B_n)$ from a given set of component invariants Φ_i.

- **Computing Component Invariants.** Invariants for atomic components are generated by simple forward analysis of their behavior. A key issue is efficient computation of such invariants as the precise symbolic computation of reachable states requires quantifier elimination. An alternative to quantifier elimination is to compute over-approximations based on syntactic analysis of the predicates occuring in guards and actions. In this case, the obtained invariants may not be inductive. D-Finder uses different strategies which allow to derive local assertions, that is, predicates attached to control locations and which are satisfied whenever the computation reaches the corresponding control location. A more detailed presentation, as well as the techniques implemented in D-Finder for generating component invariants are given in [15,16].
- **Computing Interaction Invariants.** Interaction invariants express global synchronization constraints between atomic components. Their computation consists of the following steps. 1) For given component invariants Φ_i of the atomic components B_i, we compute a finite-state abstraction $B_i^{\alpha_i}$ of B_i where α_i is the abstraction induced by the elementary predicates occuring in Φ_i. This step is necessary only for components B_i which are infinite state. 2) The system $\gamma(B_1^{\alpha_1}, \cdots, B_n^{\alpha_n})$ which is an abstraction of $\gamma(B_1, \cdots, B_n)$, can be considered as a 1-safe Petri net. The set of the traps of the Petri net defines a global invariant which we compute symbolically. 3) The concretization of this invariant gives an interaction invariant of the initial system.

2 Tool Structure

D-Finder consists of a set of modules interconnected as shown in Figure 1.

It takes as input a BIP program and progressively finds and eliminates potential deadlocks. It basically works as follows. First, it constructs the predicate

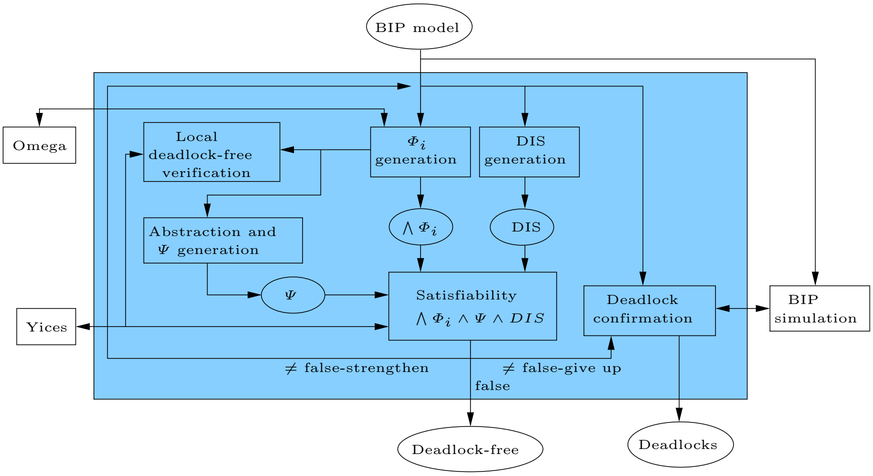

Fig. 1. D-Finder tool

characterizing the set of deadlock states (DIS generation module). Second, iteratively, it constructs increasingly stronger local invariants of components (Φ_i generation module). These invariants are used to compute finer finite state abstractions and increasingly stronger global interaction invariants (Abstraction and Ψ generation module). Third, it verifies deadlock-freedom by checking satisfiabilty of $\wedge\Phi_i \wedge \Psi \wedge DIS$ (satisfiability module). If it succeeds, the system is proven deadlock-free; otherwise it may continue or gives up, according to the user's choice.

For doing all this, D-Finder is connected with several external tools. It uses Omega [17] for quantifier elimination and Yices [18] for checking satisfiability of predicates. It is also connected to the state space exploration tool of the BIP platform, for finer analysis when the heuristic fails to prove deadlock-freedom. We provide non trivial examples showing the capabilities of D-Finder as well as the efficiency of the method.

3 Experimentation and Concluding Remarks

We provide experimental results for four examples. The first example is Utopar, an industrial case study of the European Integrated project SPEEDS[1] about an automated transportation system. A succinct description of Utopar can be found at http://www.combest.eu/home/?link=Application2. The system is the composition of three types of components: autonomous vehicles, called U-cars, a centralized Automatic Control System and Calling Units. The latter two types have (almost exclusively) discrete behavior. U-cars are equipped with a local controller, responsible for handling the U-cars sensors and performing various routing and driving computations depending on users' requests. We analyzed a simplified version of Utopar by abstracting from data exchanged between components as well as from continuous dynamics of the cars. In this version, each U-Car is modeled by a component having 7 control locations and 6 integer variables. The Automatic Control System has 3 control locations and 2 integer variables. The Calling Units have 2 control locations and no variables. In the second example, we consider Readers-Writer systems in order to evaluate how the method scales up for components without data. The third example is Gas Station in order to compare with other compositional method *assume-guarantee* [19]. Finally, as a last example, we consider Dinning Philosophers which is a well-known classical example.

The table below provides an overview of the experimental results obtained for these examples. In this table, n is the number of BIP components in the example, q is the total number of control locations, x_b (resp. x_i) is the total number of boolean (resp. integer) variables, $D_{\Phi\Psi}$ is the number of deadlock configurations remaining in $\wedge\Phi_i \wedge \Psi \wedge DIS$ and t is the total time for computing invariants and checking for satisfiability of $\wedge\Phi_i \wedge \Psi \wedge DIS$. Detailed results are available at http://www-verimag.imag.fr/˜ thnguyen/tool.

The results presented by Cobleigh and his colleagues in [19] raise doubts about the usefulness of assume-guarantee reasoning techniques. They undertook

[1] (http://www.speeds.eu.com/)

example	n	q	x_b	x_i	$D_{\Phi\Psi}$	t
Utopar System (40 U-Cars, 256 Calling Units)	297	795	40	242	0	3m46s
Utopar System (60 U-Cars, 625 Calling Units)	686	1673	60	362	0	25m29s
Readers-Writer (7000 readers)	7002	14006	0	1	0	17m27s
Readers-Writer (10000 readers)	10002	20006	0	1	0	36m10s
Gas station (100 pumps - 2000 customers)	1101	4302	0	0	0	14m06s
Gas station (300 pumps - 3000 customers)	3301	12902	0	0	0	33m02s
Philosophers (2000 Philos)	4000	10000	0	0	3	32m14s
Philosophers (3001 Philos)	6001	15005	0	0	1	54m34s

a study to determine if assume-guarantee reasoning provides an advantage over monolithic verification. In this work, they considered all two-way decomposition for a set of systems and properties, using two different verifiers, FLAVERS and LSTA. By increasing the number of repeated tasks in the systems, they evaluated the decompositions as they were scaled. They found that in only a few cases assume-guarantee reasoning can verify properties on larger systems than monolithic verification can, and in these cases the systems that can be analyzed are only a few sizes larger.

In our case, we also did some comparison with some well-known monolithic verification tools such as NewSMV (NuSMV).All the experimentations are done on a Linux machine Intel Pentium 4 3.0 GHz and 1G Ram.

The first comparison between NuSmv and D-Finder is on Dinning Philosopher example. We increase the number of Philosophers and compare the verification time between these two tools (figure 2). In the figure 2, NuSmv runs out of memory at the size 150 while D-Finder can go much further until the size 3000.

The second comparison between NuSmv and D-Finder is on Gas Station example. We consider a system with 3 pumps and increase the number of customers. The comparison of verification time is in figure 3. In this figure, NuSmv runs

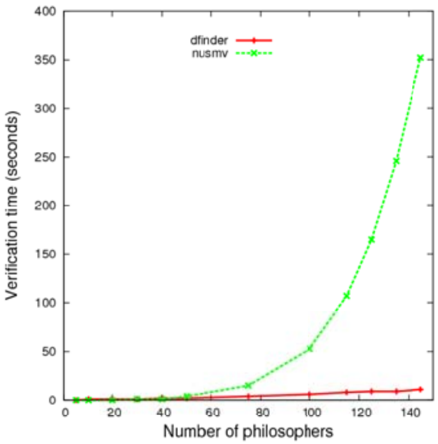

Fig. 2. Dinning Philosopher

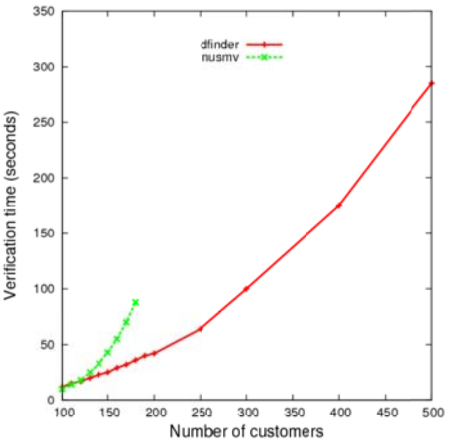

Fig. 3. Gas Station

out of memory at the size 180 (Customers) while D-Finder can go much further until the size 3000.

References

1. Alur, R., Henzinger, T.: Reactive modules. In: Proceedings of the 11th Annual Symposium on LICS, pp. 207–208. IEEE Computer Society Press, Los Alamitos (1996)
2. Abadi, M., Lamport, L.: Conjoining specifications. Toplas 17(3), 507–534 (1995)
3. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In: Proceedings of the 4th Annual Symposium on LICS, pp. 353–362 (1989)
4. Chandy, K., Misra, J.: Parallel program design: a foundation. Addison-Wesley Publishing Company, Reading (1988)
5. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
6. McMillan, K.L.: A compositional rule for hardware design refinement. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 24–35. Springer, Heidelberg (1997)
7. Pnueli, A.: In transition from global to modular temporal reasoning about programs, pp. 123–144 (1985)
8. Stark, E.W.: A proof technique for rely/guarantee properties. In: Maheshwari, S.N. (ed.) FSTTCS 1985. LNCS, vol. 206, pp. 369–391. Springer, Heidelberg (1985)
9. Kupferman, O., Vardi, M.Y.: Modular model checking. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 381–401. Springer, Heidelberg (1998)
10. Peterson, J.: Petri Net theory and the modelling of systems. Prentice Hall, Englewood Cliffs (1981)
11. Apt, K.R., Francez, N., de Roever, W.P.: A proof system for communicating sequential processes. ACM Trans. Program. Lang. Syst. 2(3), 359–385 (1980)
12. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: SEFM, pp. 3–12 (2006)
13. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 98–112. Springer, Heidelberg (2001)
14. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD, pp. 173–180 (2007)
15. Bensalem, S., Lakhnech, Y.: Automatic generation of invariants. FMSD 15(1), 75–92 (1999)
16. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional verification for component-based systems and application. In: Cha, S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 64–79. Springer, Heidelberg (2008)
17. Team, O.: The omega library. Version 1.1.0 (1996)
18. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
19. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. ACM TSEM 17(2) (2008)