D-Finder 2: Towards Efficient Correctness of Incremental Design

Saddek Bensalem[1], Andreas Griesmayer[1], Axel Legay[3], Thanh-Hung Nguyen[1], Joseph Sifakis[1], and Rongjie Yan[1,2]

[1] Verimag Laboratory, Université Joseph Fourier Grenoble, CNRS
[2] State Key Laboratory of Computer Science, Institute of Software, CAS, Beijing
[3] INRIA/IRISA, Rennes

Abstract. *D-Finder 2* is a new tool for deadlock detection in concurrent systems based on effective invariant computation to approximate the effects of interactions among modules. It is part of the BIP framework, which provides various tools centered on a component-based language for incremental design. The presented tool shares its theoretical roots with a previous implementation, but was completely rewritten to take advantage of a new version of BIP and various new results on the theory of invariant computation. The improvements are demonstrated by comparison with previous work and reports on new results on a practical case study.

1 Context

Language. *D-Finder 2* is part of a framework of tools that share a common language, *BIP*, to describe component-based systems [1]. The language is based on *atomic components* and *connectors* to describe their interactions. Components can also be hierarchically organized to build new components. An atomic component is a transition system $B = (L, P, \mathcal{T})$, where $L = \{l_1, l_2, \ldots, l_k\}$ is a set of control locations, P is a set of ports, and $\mathcal{T} \subseteq L \times P \times L$ is a set of transitions. A component additionally can contain data and use C code for actions and conditions on the transitions to manipulate this data. Figure 1 shows a graphical representation of two atomic components B_1 and B_2. We use cycles for locations and arrows for transitions. Every transition is labeled by a port to synchronize with ports of other components to create interactions. In the example, the ports *trigger* and *tick* of the two components are synchronized, which means the corresponding transitions have to be executed concurrently, and are only available if the guards in both components are fulfilled. The transition *rel* can be taken whenever a component is in the *fire* location. We can give only a very brief description of *BIP* here, please refer to [1] for more details.

Verification. Previous work [3,4] introduced an efficient verification method for the models above. Key to this method is the approximation of the reachable states by compositional invariant computation based on (1) *component invariants* Φ_i that capture the constraints on local data of a component B_i, and (2)

M. Bobaru et al. (Eds.): NFM 2011, LNCS 6617, pp. 453–458, 2011.
© Springer-Verlag Berlin Heidelberg 2011

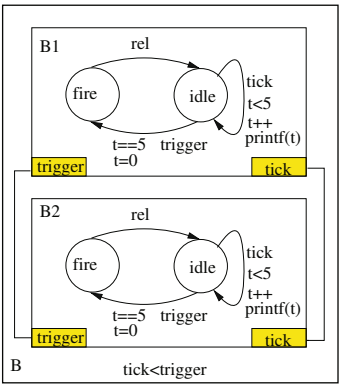

Fig. 1. A BIP model

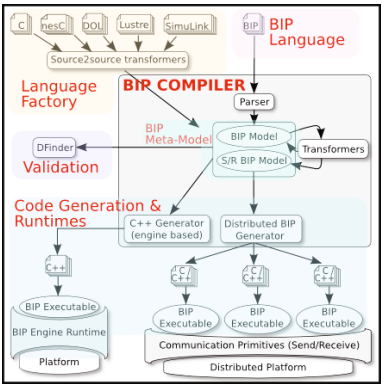

Fig. 2. BIP tools and work-flow

the *interaction invariant* Ψ, which captures constraints on the global state space induced by the synchronization. More formally, we have the following rule:

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

The rule states that if all components B_i fulfill their respective *component invariants* Φ_i, the composition of all components $II(\|_\gamma\{B_i\}_i, \{\Phi_i\}_i)$ with the interactions γ fulfills an *interaction invariant* Ψ, and if furthermore the conjunction of the invariants $(\bigwedge_i \Phi_i) \wedge \Psi$ implies a predicate on the global system Φ, then also the global system $\|_\gamma\{B_i\}_i$ itself fulfills Φ. In this paper we concentrate on global deadlock-freedom. Indeed, it suffices to prove the invariance of the predicate $\neg\mathcal{DIS}$, where $\mathcal{DIS}$ is the set of states of the system from which all the interactions are disabled.

Tool Chain. The design flow between *BIP* and *D-Finder* is sketched in Figure 2. The framework allows to (1) start from scratch and describe a composite system with the BIP language, or (2) to use the *Language Factory* to translate existing models described in languages such as C, DOL [15] or Simulink [13] into the BIP framework. These models then are used for validation, verification, model to model transformation and eventually generation of C++ code for simulation or deployment. *D-Finder* plays a central role in this process to verify the initial models as well as ensuring correctness after transformation steps.

2 *D-Finder 2*

Recently, *BIP* has been updated and enriched with new features to improve the modeling process for building hierarchical models and add new interactions in an incremental manner. Furthermore, since the tool presentation in [4], new, more efficient techniques for computing Ψ were introduced in [6,2]. To show the results of unifying those recent developments, this paper presents *D-Finder 2*, the

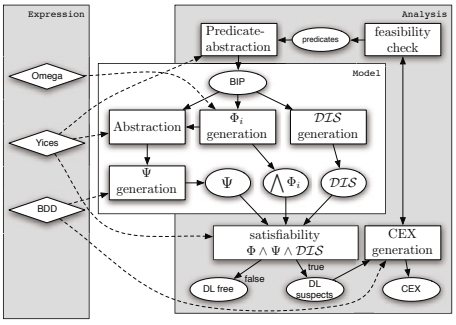

Fig. 3. Structure of the D-Finder tool

```
#> dfinder -f p1000.bip --incr_file incr_15.incr
   --method=pm --analysis=dl
#  overall analysis                              :
#    compute II using incremental pm             :
#      Eliminate Variables Abstraction(Phil...   :
#        Compute CI for Philosopher              : 0:01
#      Eliminate Variables Abstraction(Phil...   : 0:02
...
#      get common locations                      : 0:03
#      compute BBCs[0]                           : 0:01
...
#      integrate for increment[1]                : 0:00
...
#      dual compuatation                         : 0:00
#      concretization                            : 0:02
#    compute II using incremental pm             : 0:41
#    incremental DIS                             : 0:24
     Found 1 deadlocks:
#  overall analysis                              : 1:07
```

Fig. 4. Call from the command line

second edition of the *D-Finder* tool-set. The tool has been entirely rewritten and new techniques for computing invariants have been implemented in a modular manner.

2.1 Computing Interaction Invariants in an Incremental Manner: the Theory behind *D-Finder 2*

D-Finder 2 implements new efficient techniques for computing Ψ that were recently introduced in [6,2]. Those techniques build on the new concept of *Boolean Behavioral Constraints* (BBCs) that allow to relate the communication between different components with their internal transitions and hence model a unified invariant of the model. Solutions of BBCs can be used to symbolically compute a strong interaction invariant. There are two different techniques that exploit BBCs: (1) a symbolic computation based on a *Fixed-Point iteration* (FP), and (2) a symbolic algorithm to solve the BBCs using so called *Positive Mapping* (PM). Both methods allow an efficient implementation for computing interaction invariants using BDDs and show their strengths for different topologies of the model to check. The main advantage of the two aforementioned techniques is that they allow to exploit the component based design of *BIP* and compute interaction invariants incrementally. In the *Incremental Fixed Point* (IFP) and *Incremental Positive Mapping* (IPM) methods, *D-Finder 2* partitions the model into subsystems (also called increments). The internal interactions in these subsystems are used to compute "partial" interaction invariants. Relations between different increments are considered in a second step and used to integrate the intermediate results to the final Ψ. Computing the global interaction invariant from smaller intermediate results allows to reduce the size of the data structures involved in the computation.

2.2 Implementation Details

D-Finder 2 was developed with modularity and extensibility in mind. The tool is written in Java and uses external tools and native code via the Java Native Interface (JNI) for computations. Fig. 3 gives an overview of the main modules

of the tool. The *Model* block handles the parsing of the *BIP* code into an internal model and provides the means to compute Φ, Ψ, and $\mathcal{DIS}$. Available implementations comprise the methods from the previous tool and additionally the new algorithms using *fixed-point* and *positive mapping* computation and their incremental versions. The results from *Model* are used from various implementations of the *Analysis* block, which perform further steps like the generation of possible deadlocks and, most recently, generation of counterexamples for Boolean systems (CEX). The *Expression* block is used by both *Model* and *Analysis*. Its main purpose is to provide an uniform interface for different back-ends that store the actual expressions. The abstraction from the back-ends allows a high degree of flexibility for implementing the algorithms. The most general implementation is a wrapper to the actual parse tree representation of the expressions (using the Eclipse Modeling Framework, EMF) and uses external tools for computations. For algorithms on Boolean variables, like computation of Ψ, a more succinct implementation with BDDs as back-end is used, while large systems that incorporate non-Boolean data require to directly create and maintain input files for an SMT solver on disk. These different versions of expressions can be used interchangeably in many contexts, with the respective tools being called transparently for actual computations. The *Expression* block also provides methods to translate between representation and manage the scopes of variables. Fig. 3 shows the use of external tools for models with non-Boolean data; for models with only Boolean variables, BDDs are used in all computation steps.

The three main blocks are complemented by a common configuration module that reads settings from default values, configuration files and command line and provides the means to instantiate the proper modules for an example to check. The used tools for SMT solving (Yices, [16]) and variable quantification (omega library, [12] are accessed using wrappers, giving rise for easy extension and replacement. Similarly, the used BDD-Manager (JavaBDD, [11]) provides a Java implementation, but has the option to use native BDD managers on supported machines. Currently we use CUDD [14] on Linux and OS X. This flexibility provides the means to develop and maintain new and experimental algorithms in the tool while leaving the main behavior intact, which is currently done, e.g., for experimental modules to perform predicate abstraction to create Boolean systems, check the reachability of deadlocks to remove false positives, and to construct error traces to understand the causes of reachable deadlocks, all of which were not present in the previous tool.

2.3 Availability of the Tool and Example of Use

The *D-Finder 2* and *BIP* tools, along with the examples discussed in this paper, can be freely downloaded from [9] and [7] respectively. An excerpt of a call to *D-Finder 2* for the case of dining philosophers is given in Figure 4. The example shows verification of a problem size of 1000 Philosophers partitioned into 20 increments. The first step is the computation of an abstraction to remove the variables for Ψ computation (using the post conditions from Φ computation to split the states), followed by the local computations (BBC) for each of the

Table 1. Comparison of D-Finder versions. Times in min, timeout one hour.

System Information				D-Finder 1	D-Finder 2			
scale	comps	locs	intrs	*Enum*	*PM*	*FP*	*IPM*	*IFP*
Dining Philosopher								
100 philos	200	600	500	0:06	0:09	-	0:03	0:21
500 philos	1000	3000	2500	1:51	3:32	-	0:22	3:09
1000 philos	2000	6000	5000	7:08	14:57	-	0:50	19:05
1500 philos	3000	9000	7500	19:30	34:23	-	1:34	-
3000 philos	6000	18000	15000	-	-	-	4:57	-
Gas Station								
300 pumps	3301	12902	12000	33:02	36:01	11:32	2:03	4:18
400 pumps	4401	17202	16000	-	-	21:40	3:41	10:30
500 pumps	5501	21502	20000	-	-	-	5:48	20:05
ATM System								
2 atms	6	48	38	0:59	0:05	0:02	0:02	0:02
20 atms	42	444	362	-	1:12	1:00	0:43	1:13
50 atms	102	1104	902	-	7:14	8:00	1:57	11:22
100 atms	202	2204	1802	-	-	-	4:60	-
200 atms	402	4404	3602	-	-	-	17:07	-

Table 2. Verification times for the Dala robot

module	comps	locs	intrs	vars	time	
					D-Finder 1	D-Finder 2
RFLEX	56	308	227	35	9:39	3:07
NDD	27	152	117	27	8:16	1:15
SICK	43	213	202	29	1:22	1:04
Aspect	29	160	117	21	0:39	0:21
Antenna	20	97	73	23	0:14	0:13
Combined	198	926	724	132	-	5:05

increments and their integration. Computation of the dual and mapping to the concrete values finishes the computation of Ψ, which is used to directly compute the intersection with $\mathcal{DIS}$. Finally, the tool successfully reports one deadlock.

Large examples from real world applications may require manual assumptions on the components to rule out false positives. *D-Finder 2* supports these additional inputs on the component and global level. To support organization of the required models, specifications, and output files, the tool supports so called example configuration files, which allow to collect the required files in own directories. The examples and case studies on the web site are organized in this way. The Web page of *BIP* [7] gives more information to introduce the language as well as details on usage and case studies. The web page of *D-Finder 2* [9] comes up with illustrations of the use of the tool as well as many other case studies of huge size. The sites also reference a series of publications that give more details on the theory implemented in the tool.

3 Experimental Results

We compare the performance of the original version of *D-Finder* with the new version of the tool presented in this paper on some case studies (see [9] for more experiments). Experiments where conducted with a 32Bit Linux on Xeon 2.67GHz We started by considering verification of deadlock properties for the classical case studies of Dining Philosopher, the Gas Station [10], for which we assume that every pump has 10 customers, and the Automatic Teller Machine (ATM) [8]. The results are given in Table 1, where *scale* is the parameter of the example, *comps* the number of components, *locs* the number of control locations, and *intrs* the total number of interactions. The experiments where performed on Mac-Book Pro laptops with CUDD as back end for BDD computations. We see that especially the incremental versions of the new Ψ computation methods led to major improvements in run time compared to the original version of the tool from [4].

To demonstrate the application of *D-Finder 2* to industrial problems we want to refer to a case study on the application of *BIP* to autonomous robots *Dala* [5].

The case study uses software written in *Genom*, a tool to design modular real time software architectures, which were then translated to *BIP*. The translated modules implement features of the robot like movement (RFLEX), navigation (NDD) and self localization using a laser range finder (SICK), and themselves consist of more internal components, for more details see [5]. The case study shows how to use *BIP* code generation tools and the *BIP engine* to create C code from the model, which runs at the functional level of the robot to guarantee coordination of the various modules in a correct manner. The previous tool was not able to verify this model. And while prototypes were able to show the deadlock-freedom of single modules in the past, only *D-Finder 2* allowed us recently to verify the combination of all five main components. (Results are reported in Table 2). This use of the work flow of *D-Finder 2* and *BIP* is a major change with respect to other methodologies to design autonomous robots. Indeed, most of other existing works propose functional levels that are designed manually, without any formal guarantee of correctness.

References

1. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM, Washington, DC, USA, pp. 3–12. IEEE, Los Alamitos (2006)
2. Bensalem, S., Bogza, M., Legay, A., Nguyen, T.-H., Sifakis, J., Yan, R.: Incremental component-based construction and verification using invariants. In: FMCAD (2010)
3. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: Compositional verification for component-based systems and application. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 64–79. Springer, Heidelberg (2008)
4. Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J.: D-finder: A tool for compositional deadlock detection and verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 614–619. Springer, Heidelberg (2009)
5. Bensalem, S., de Silva, L., Gallien, M., Ingrand, F., Yan, R.: Rock solid software: A verifiable and correct by construction controller for rover and spacecraft functional layers. In: ISAIRAS, pp. 859–866 (2010)
6. Bensalem, S., Legay, A., Nguyen, T.-H., Sifakis, J., Yan, R.: Incremental invariant generation for compositional design. In: TASE, pp. 157–167 (2010)
7. BIP tool page, `http://www-verimag.imag.fr/BIP-Tools,93.html`
8. Chaudron, M.R.V., Eskenazi, E.M., Fioukov, A.V., Hammer, D.K.: A framework for formal component-based software architecting. In: SVCS (2001)
9. DFinder tool page, `http://www-verimag.imag.fr/dfinder/`
10. Heimbold, D., Luckham, D.: Debugging Ada tasking programs. IEEE Softw. 2(2), 47–57 (1985)
11. JavaBDD tool page, `http://javabdd.sourceforge.net/`
12. Omega library tool page, `http://www.cs.umd.edu/projects/omega/`
13. Simulink, `http://www.mathworks.com/products/simulink/`
14. Somenzi, F.: CUDD tool page, `http://vlsi.colorado.edu/~fabio/CUDD/`
15. Thiele, L., Bacivarov, I., Haid, W., Huang, K.: Mapping applications to tiled multiprocessor embedded systems. In: ACSD, pp. 29–40. IEEE, Los Alamitos (2007)
16. Yices tool page, `http://yices.csl.sri.com/`