

# On Distributed Verification of Generalized Interaction Models of Software Components<sup>\*</sup>

Viliam Holub

Distributed Systems Research Group  
Faculty of Mathematics and Physics, Charles University in Prague

**Abstract.** The presented work aims at developing a distributed verification tool for a generalized automata system. The verification is done by an exhaustive state-space exploration – a stream of linearly-organized system states is updated on each node of a computational cluster and handed over to the next node over a network. The automata system is proposed to be general enough to describe common automata-based formalisms used for describing interaction of software components. We also present results obtained with the proof-of-concept implementation.

## 1 Motivation and Problem Statement

Model checking of software components is an approach to automatically verify selected aspects of component correctness. As software itself is very difficult to verify, abstracted models and constraint languages have been introduced. We focus on those models and languages, that are based on automata and interactions among them, such as the specification of the transition rules and component composition. In behavior protocols[1] these models and languages allow us to check rules and bounds on interface utilization, and even to check complex interactions in the entire component system based on a complete description of possible actions and reactions.

The existing approaches to model-checking of complex software architectures are trying to overcome two major problems. One is the problem inherent to the complexity of the model – the number of states of a model tends to grow exponentially with the size of the model, leading to a situation where the model either does not fit into the available memory, or the time to visit the states of the model is too long. This is referred to as a state explosion. The other is the problem inherent to the complexity of the model-checking methods – the methods consist of complex algorithms that are often implemented as a proof of concept only, thus lacking the potential an optimized implementation would bring.

One of the approaches to solve the state space explosion problem is distribution of the model checking algorithm. The typical method is dividing the state space into distinct parts[2], where each node checks only the assigned states and, if necessary, asks the other nodes to visit other reachable states as the model

---

<sup>\*</sup> This work was supported by the Czech Science Foundation (GACR) 201/05/H014.

checking algorithm traverses the state space. The network latency and load becomes a bottleneck, giving rise to a hard problem of optimal state distribution. Moreover, due to random access to states, the state space cannot be effectively saved on mass storage devices.

Our goal is to solve these problems, namely:

- To develop an effective distributed verification tool based on exhaustive exploration of linearly-organized state space and error state reporting. We restrict the problem to systems that describe interactions among components, are based on Label Transition Systems (LTS), and have a finite number of states. Because of limitations on the model, we can use more powerful optimizations than similar systems based on complex formalisms (LTL, CTL, models with unlimited state space).
- To design a formal model of component interaction should the existing models prove unsuitable. The model should be general enough to allow describing a majority of common constructs available on LTS and be suitable for effective automated verification.
- To present the approaches to conversion from other formal models to the verified one and to show how to interpret the results.

## 2 Related Work and Model Analysis

The related work discusses ADL-based approaches to modeling, the abstract modeling approaches based on automata, and the problem of model verification.

The Wright Architectural Description Language[3] uses a process algebra (a subset of CSP) for defining component behavior. The tested property is called compatibility[4] and in short says that the connector interaction cannot detect that the role process has been replaced by the port process. Behavior protocols[1] (BP), embedded in SOFA[5], allow the simplification of the verification process by hierarchical decomposition of the architecture, starting with the smallest parts of the system and continuing in the bottom-up manner, thus reducing the internal complexity of composite components. BP allows identifying several composition errors, including no activity (deadlock) and bad activity. Furthermore, it defines compliance[6] as the relation of expected and real behavior.

Interface automata[7] allow compatibility checks between interface models in optimistic view, i.e. components can be used together if there is at least one correct design. The composition is defined on two automata only, synchronizing on one input and one output action. Team automata[8] is a composition of component automata. The specified action is executed in the team by simultaneously executing the action in all component automata. There is no limitation for the number of states. Component-interaction automata[9] has been developed with respect to ADL and software components. The concept is similar to team automata; however there are differences the authors give reasons to be more suitable for software design.

Although a large spectrum of formalisms has been mentioned and analyzed, none of them is both general enough and easy to verify. Thus, we have decided

to propose a new model denoted later in the text as *interference automata*. It is close to team automata.

Notably, interference automata are not limited to one type of synchronization (synchronous, asynchronous or blocking, non-blocking) nor a number of action participants (one-to-one, one-to-many, and many-to-many). Having this in mind, the model can be outlined as a set of local automata, augmented by a set of transition rules and a set of alarms. A system state is a vector of states of all the local automata. A transition is a pair of starting and final state. For practical reasons, a range of starting and final states is allowed as well. Alarms describes states which are somehow “interesting” and its reachability should be reported (often it is related to erroneous states). The proposed model does not cover all the needs of the mentioned formal methods (for example, the unlimited number of states), but fits perfectly to our requirements.

There are many verification tools (including SPIN[10] and DiVinE[11]) that aim mainly at checking LTL or CTL formula. SPIN uses the Depth First Search (DFS) to explore the state space and uses the operational memory as a state cache to prevent a re-exploration of previously visited states. Because a random access to data structures is required, external slow devices such as hard-disks cannot be used effectively. We believe that targeting simpler interaction protocols and relations among components allows for a more efficient verifier implementation.

### 3 Distributed Verification

We face the need of storing a large amount of states via a special state space encoding. All properties (such as *state-explored* and *state-to-be-explored*) of reached states are stored sequentially in the order of the DFS traversal algorithm, creating a stream. Because a numeric value of the state can be computed from its position in the stream, it is omitted. This approach saves a significant amount of storage space. The stream can be passed across computational nodes organized in a logical circle, allowing all the nodes to work simultaneously on different parts of the stream and systematically updating the state space. Because of the linear access pattern, buffering on mass storage devices is possible with nearly zero-overhead.

A drawback of this approach is the need to avoid random access to the stream. During the verification process, nodes access the states only sequentially and use their operational memory as a state cache and a state buffer. In the case of memory overflow, the node has to suspend the exploration, store the explored states from the state buffer, and thus free the sufficient amount of memory.

Because forced flushing of the state buffer reduces verification effectivity, three basic techniques to reduce memory consumption are used. The first technique is a static optimization which affects each component of the system individually (automata minimalization). A dynamic optimization takes into account the interactions among automata, trying to identify unreachable system states and remove them from the model. This improves the state space compression.

The third optimization is applied on-the-fly during the verification process. A coherent part of the explored state space with the same properties is replaced by a single cut-off superstate.

We plan to prove the concept on a real-life project[12]. At present, a basic, single-node prototype without dynamic optimizations shows promising results. For instance, the average amount of explored states is about 5 millions states per second (sps) on a low-end desktop, compared to 180000 sps of the SPIN model checker. This will change as more optimizations will be included. On a testing example with  $10^7$  states, the overall memory requirements were 6 bits per state.

Future work will focus on implementation of a parallel reduction technique (with less overhead than classical LTL and CTL verifiers have) and further static model optimizations.

## References

1. Plasil, F., Visnovsky, S., Besta, M.: Bounding component behavior via protocols. In: TOOLS. Volume 30., USA, IEEE Computer Society (1999) 387–398
2. Garavel, H., Mateescu, R., Smarandache, I.: Parallel state space construction for model-checking. In: SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software, New York, NY, USA, Springer-Verlag (2001) 217–234
3. Allen, R.J.: A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon University (1997) Chair-David Garlan.
4. Allen, R., Garlan, D.: A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **6** (1997) 213–249
5. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: International Conference on Configurable Distributed Systems (CDS), Washington, DC, USA, IEEE Computer Society (1998)
6. Adamek, J., Plasil, F.: Erroneous architecture is a relative concept. In: Software Engineering and Applications (SEA) conference, Cambridge, MA, USA, ACTA Press (2004) 715–720
7. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Ninth Annual Symposium on Foundations of Software Engineering (FSE), New York, NY, USA, ACM Press (2001) 109–120
8. Ellis, C.: Team automata for groupware systems. In Hayne, S., Prinz, W., eds.: SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP), ACM Press, New York (1997) 415–424
9. Zimmerova, B., Brim, L., Cerna, I., Varekova, P.: Component-interaction automata as a verification-oriented component-based system specification. *SIGSOFT Software Engineering Notes* **31** (2006)
10. Holzmann, G.J.: The model checker SPIN. *IEEE Transactions on Software Engineering (TSE)* **23** (1997) 279–295
11. Barnat, J., Brim, L., Cerna, I., Simecek, P.: DiVinE the distributed verification environment. In Leucker, M., van de Pol, J., eds.: 4th International Workshop on Parallel and Distributed Methods in verification (PDMC), Lisbon, Portuga (2005)
12. Jezek, P., Kofron, J., Plasil, F.: Model checking of component behavior specification: A real life experience. In: International Workshop on Formal Aspects of Component Software (FACS'05). (2005)