Session-Based Compositional Verification on Actor-based Concurrent Systems *

Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen

Department of Computer Science, Technische Universität Darmstadt, Germany
`eduard.kamburjan@gmx.de, crystald@cs.tu-darmstadt.de, tc.chen@dsp.tu-darmstadt.de`

Motivations Concurrent and distributed systems are the pillars of modern IT infrastructures. It is of great importance that such systems work properly. However, quality assurance of such systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. Besides, it is non-trivial to ensure communication (composed by interactions) safety: as developers implement applications locally, the sending application's expected sequence of interactions may not fit the receiving application's. Thus interactions between endpoints could become inconsistent and unexpected messages may damage endpoint applications such that extra cost is caused to the overall system. These challenges motivate compositional frameworks combining precise modeling and analysis with suitable tool support for such kind of systems. In particular, it is crucial to provide a verification framework which is able to analyze the overall interactions and structurally verify endpoint applications in an intuitive way with respect to endpoint behavior.

Object orientation is the leading framework for concurrent and distributed systems. Concurrent objects combine object-orientation with the *actor* model [7]. Actors communicate with one another by asynchronous message passing, which allows the caller to continue with its own activity without blocking while waiting for the reply. Moreover, the notion of *futures* [6] improves this paradigm by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing future identities, the caller enables other objects to wait for the same method results.

The Proposed Framework In this work, we take *ABS* [5] as a modeling language for concurrent and distributed systems. *ABS* is based on Creol [10]. It is imperative, object-oriented, executable and it supports concurrent objects and shared futures.

The observable behavior of a system can be described by *communication histories* over observable events [8]. Due to asynchronous message passing in *ABS*, [3, 4] propose a disjoint event semantics, in which events are separated for method invocation, reacting upon a method call, resolving a future, and for fetching the value from a future. Each event is observable to only one object, which is the one that generates the event.

The theorem prover KeY-ABS [2] based on KeY [1] is developed for verifying history-based class invariants for *ABS* models. The class invariants can (1) relate the internal object states with the interactions between the current object and the surrounding environment, or (2) express the structure of histories local to the current object. The proof rule for compositional reasoning about *ABS* programs is given and proved sound in [4], by which system invariants can be obtained from the class invariants proved by KeY-ABS through history composition. This bottom-up verification approach by KeY-ABS is based on relay-guarantee mechanism for each class in the model. In this work, we propose a top-down verification approach to verify the overall behavior between concurrent and distributed endpoints.

*The work has been supported by the EU project FP7-610582 *Envisage: Engineering Virtualized Services* (`http://www.envisage-project.eu`).

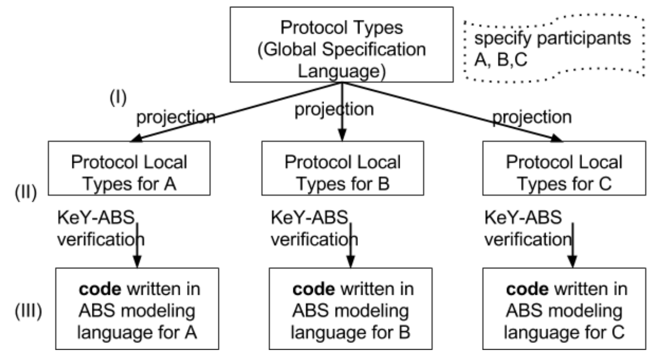
Protocol Types (Global Specification Language)
specify participants A, B,C
(I)
projection
projection
projection
Protocol Local Types for A
Protocol Local Types for B
Protocol Local Types for C
(II)
KeY-ABS verification
KeY-ABS verification
KeY-ABS verification
(III)
code written in ABS modeling language for A
code written in ABS modeling language for B
code written in ABS modeling language for C

Figure 1: Session-Based Compositional Verification Framework

Session types [9] establish a means of typing concurrent and asynchronous interactions among distributed components. In this work, we propose a session-based verification framework for concurrent and distributed *ABS* models. We type applications' behaviors, which include the usage of *future*, with respect to sessions where the applications are participating in, and partition those behaviors based on sessions. We call the extended session types as *protocol types*, which not only enjoy all features defined in session types, but also specify the timing for invoking feature. Let $\mathsf{p}, \mathsf{q}, ..$ range over endpoint process identifiers, let l be labels for options in a branching, and let f be future identities. The syntax of protocol types is defined below:

$$\begin{array}{rrcl}
\text{(Sorts)} & S & ::= & T \mid \langle G \rangle \\
\text{(Data types)} & T & ::= & \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{string} \mid \mathit{Fut}\langle T \rangle \\
\text{(Protocol Types)} & G & ::= & \mathsf{p} \xrightarrow{f_j} \mathsf{q} : \{l_j(T_j).G_j\}_{j \in J} \mid \mathsf{Rel}(f) \mid G \parallel G \mid \mathsf{t} \mid \mu\mathsf{t}.G \mid \mathtt{end} \\
\text{(Protocol Local Types)} & L & ::= & \mathsf{q}!_{f_j}\{l_j(T_j).L_j\}_{j \in J} \mid \mathsf{p}?_{f_j}\{l_j(T_j).L_j\}_{j \in J} \mid \mathsf{Rel}(f) \mid \mathsf{t} \mid \mu\mathsf{t}.L \mid \mathtt{end}
\end{array}$$

Sorts, denoted by S, range over data types, and $\langle G \rangle$, a closed protocol type (i.e. having no type variable) of a content. This implies that we can deliver a behavior (typed by G) from one endpoint to another. Data types, T, include standard value types and future types, $\mathit{Fut}\langle T \rangle$, for the types of method parameters and method return results. A protocol type $\mathsf{p} \xrightarrow{f_j} \mathsf{q} : \{l_j(T).G_j\}_{j \in J}$ globally describes an interaction behavior in which an endpoint process p sends a content of type T_k to another endpoint process q, where f_j is a future identity and label $l_k \in \{l_j\}_{j \in J}$, $k \in J = \{1..n\}$. Then the global behavior continues with G_k. If $J = \emptyset$, then conventially we write $\mathsf{p} \xrightarrow{f_j} \mathsf{q} : (T).G$ to represent a simple sending and receiving interaction. In the protocol *local* types, $\mathsf{q}!_{f_j}\{l_j(T_j).L_j\}_{j \in J}$ and $\mathsf{p}?_{f_j}\{l_j(T_j).L_j\}_{j \in J}$ are corresponding to the interacting endpoints' behaviors defined in $\mathsf{p} \xrightarrow{f_j} \mathsf{q} : \{l_j(T_j).G_j\}_{j \in J}$. The former types the sender's behavior to send a message to q, while the later types the receiver's behavior to receive a message from p. The new type $\mathsf{Rel}(f)$ captures the process release point upon waiting for the future f to be resolved, i.e. containing method results.

For example, we can write

$$\mathsf{p}_1 \xrightarrow{f_1} \mathsf{q}_1 : \{\mathit{BigD}(\mathit{string}).\mathsf{Rel}(f_1).\mathsf{p}_2 \xrightarrow{f_2} \mathsf{q}_1 : (\mathit{unit}).\mathtt{end},\ \mathit{SmallD}(\mathit{bool}).\mathsf{q}_1 \xrightarrow{f_1} \mathsf{p}_1 : (\mathit{int}).\mathtt{end}\}$$

to globally describe the following interactional features between p_1, q_1, and p_2: there are two branches for the first interaction $\mathsf{p}_1 \xrightarrow{f_1} \mathsf{q}_1$. Branch *BigD* leads to compute a very big data at q_1

(requested by $\mathtt{p}_1$), while branch *SmallD* is for computing a small data. $\mathtt{q}_1 \xrightarrow{f_1} \mathtt{p}_1 : (int)$ implies that $\mathtt{p}_1$ will need to wait for the response from $\mathtt{q}_1$ before proceeding to the next action `end` (i.e. terminate). Since it is just a small data, $\mathtt{p}_1$ should not wait for a long time. However, for a very big data, $\mathtt{p}_1$ may waste lots of time for the response. Thus type $\mathsf{Rel}(f_1)$ is used in branch *BigD* to specify that $\mathtt{p}_1$ encounters a process release point and can proceed other actions while waiting for f_1 to be resolved. It further specifies that, only when future f_1 has been resolved, the whole global behavior can go to the next interaction $\mathtt{p}_2 \xrightarrow{f_2} \mathtt{q}_1 : (unit)$. We use $G \parallel G$ for parallel composition, and t for type variable, and $\mu\mathsf{t}.G$ for a recursive type, where every t in the recursion body G is guarded by prefixes (i.e. contractive). Other terms for local types can be similarly explained.

Concluding Remarks and Future Works The main contributions in this work include (1) protocol types are extended by adding terms suitable for capturing the notion of *futures*, (2) the communication between different *ABS* endpoints, grouped by sessions, can be captured in protocol types and verified by the corresponding session-based composition verification framework, and (3) the local protocol types, projected from protocol types, of each endpoints can be translated and reformulated into history-based class invariants for KeY-ABS, see Figure 1. Based on this achievement, we also expect to extend the verification framework for *ABS* exception handling [11].

References

[1] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[2] C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195 of *LNCS*, pages 517–526. Springer International Publishing, 2015.

[3] C. C. Din and O. Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *JLAMP*, 83(5–6):360–383, 2014.

[4] C. C. Din and O. Owe. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing*, 27(3):551–572, 2015.

[5] R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In *Formal Methods for Components and Objects*, volume 7866 of *LNCS*, pages 1–37. Springer, 2013.

[6] R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, Oct. 1985.

[7] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[8] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.

[9] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284, Jan. 2008.

[10] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[11] I. Lanese, M. Lienhardt, M. Bravetti, E. Johnsen, R. Schlatte, V. Stolz, and G. Zavattaro. Fault model design space for cooperative concurrency. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *LNCS*, pages 22–36. Springer Berlin Heidelberg, 2014.