2012 International Conference on System Engineering and Modeling (ICSEM 2012)
IPCSIT vol. 34 (2012) © (2012) IACSIT Press, Singapore

Verifying Semantic of System Composition for an Aspect-Oriented Approach

Paniti Netinant

School of Science and Technology, Information Technology Department,
Bangkok University, Thailand

Abstract. Given initial decomposition system properties of a distributed program into processes we are faced with interprocesses and intraprocesses properties that are tightly bounded. This paper concentrates on the challenges of expressing and imposing interprocesses system properties during an execution. The approach is based on two completely independent areas of research. The Communication Closed Layers (CCL) is a formal system for developing, maintaining, and verifying distributed program. The Aspect-Oriented Software Development (AOSD) enables separations of system properties of a distributed program.

Keywords: Aspect - Oriented, Distributed Programming and Formal Systems.

1. Introduction

The intra-processes properties are relationships and requirements over local state of a process. The interprocesses properties are relationships and requirements over different local processes states that depict the consistencies and cooperation among a collection of cooperative processes. Both classes of system properties are crucial for system development and verification. The intra-process properties are relatively easier to define and carry on through system development life cycle.

The approach taken here is based on two completely independent areas of research. The first one is the Communication Closed Layers (CCL) which a formal system for developing, maintaining, and verifying distributed programs [3]. The second approach is the Aspect-Oriented Software Development (AOSD) that enables simplification and automation of CCL implementations. Aspect oriented approach provides a natural framework to address interprocess cooperation. The essence of aspect-oriented software development is the localization of cross-cutting concerns. Under CCL constrains cooperative global properties can be considered as a special case of crosscutting concerns.

Both CCL and AOSD are software development methodologies. Their synergy for distributed system development and runtime verification of virtual global system assertions is a promising match to handle the complexity of global system cooperation and consistencies. CCL and AOSD are centered on the notion of crosscutting concerns. CCL captures the semantics whereas AOSD provides the expressive tools. Together, it makes a design-by-contract discipline applicable to a wider range of distributed programs.

2. System Decomposition

Decomposition of a distributed system into processes provides only a syntactic representation of one dimension: the vertical dimension. No syntactic representation is available for the decomposition of the systems into its logical phases in terms of system's goals. For example, it is not syntactically visible when the system as a whole has achieved its first sub goal and is ready to launch into the second sub-goal. At any given time, different processes might be executing at different sub-goals. The tyranny decomposition into processes overlooks the logical structure of the system as a whole.

The first attempt to break this tyranny of process decomposition is to impose complementary horizontal decomposition. Artificial barrier may be introduced to gather and hold processes together at a certain point

before allowing them to continue execution. Each process may have a set of halting points at which it suspends its execution. When all processes have reached a local halting point and the whole system halts, a global relationship among the different processes states is verified for consistency. Only if the system is on "the correct" milestone step, it allows to continue. Also, this global checking point can be used for intelligent decisions regarding system future sub-goals. Now both decompositions are syntactically visible, global invariants could be inserted and the complementary structure of the system as a sequence of sub-goals is observable. The huge benefit from such capabilities can be majored by their use in sequential, non-distributed systems. Preconditions and postconditions use assertions and invariants during sequential life cycle process.

Obviously, this approach is far from practical. Over synchronization slows the system performance more than necessary. In many cases, just the process of detecting that distributed processes have all reached a halting point is, in the best case, a hazard and in many cases just impossible. Breaking the tyranny of composition of processes required a more sophisticated approach. The idea behind the CCL is to find a reasonable set of restrictions under which processes do not have to really halt at their local halting points and yet the semantics of the program as a whole is as if they do. The horizontal decomposition of the system into its logical phases is observable and yet non-imposing in terms of execution. The proof of such semantic equivalence is given in [3].

In [3] presented a synergy of object-oriented distributed programming and CCL design by con-tract for distributed applications where processes are tightly bounded to achieve a unique common goal. The idea is that object orientation provides encapsulation of the communication among processes and hence enables a syntactic identification of a local process unit that collaborates with similar units in other processes. CCL enables the syntactic identification of such cooperation. The actual implementation of layer boundaries is inserted into each process code. This result with the well known tangling code phenomenon of crosscutting concerns and here is where aspect orientation can provide a natural solution.

The integration between CCL and aspect orientation for distributed software development is simple; local halting points for each process would be realized by aspect oriented approach called *joint points*. *Pointcut* designators will filter out a subset of all the joint points in different processes for the identification of a virtual global state that represent the logic behind the horizontal structure of the program. *Before advice*, *after advice*, *around advice*, and *aspects* would have their natural role. The complexity results from a massy code tangling in the *chessboard* implementation are removed. The system acquires a higher degree of flexibility and adaptability associated with aspect-oriented design.

The Communication Closed Layer (CCL) is a language independent methodology that is represented only in terms of semantic constraints among its components. Similarly, this paper will present the CCL realization independent of any particular aspect oriented technology.

3. Layers

The notion of CCL is an independent language. A complete formal definition and proof system can be found in [3, 7]. More work on CCL can be found in [4, 6, 7, 8, 10, 14, 15]. This paper will present only a general, more intuitive definition using CSP language notation.

Note that this representation reflects the tyranny of processes decomposition. Now assume that all the *j* segments are communicating to achieve the overall system's sub-goal. We would like this fact to be syntactically represented.

$[S_{1j} \parallel S_{2j} \parallel \ldots \parallel S_{nj}]$ is called the *j* layer of the system $L_j :: [S_{1j} \parallel S_{2j} \parallel \ldots \parallel S_{nj}]$. We would like to use layers representation to compose the whole system back. There are two different composition rules: the Sequential Composition

Rule (SCR) and the Distributed Composition Rule (DCR).

Ideally, we would like to use SCR through the software life cycle but use the DCR at runtime. The problem is that these two compositions, in general, do not yield the same semantics. The SCR program exhibits only a subset of all possible computation paths that can occur in the DCR program. This means that,

in general, the transformation from SCR to DCR at runtime does not necessarily preserve the program semantics.

Let [P_1 || P_2 || … || P_n] be a distributed program **P** composed of n processes P_1 to P_n.

Now assume each of these processes is decomposed into its logical segments. Each P_i is refined into:

begin S_{i1}; S_{i2}; … ; S_{ik} end.

S_{ij} is the *j* segment of processes i. The distributed program can be expressed as:

$$
\begin{array}{c}
\text{begin } S_{11};S_{12}; \dots ;S_{1k} \text{ end} \\
\| \\
\text{begin } S_{21};S_{22}; \dots ;S_{1k} \text{ end} \\
\| \\
\text{begin } S_{31};S_{32}; \dots ;S_{1k} \text{ end} \\
\| \\
\dots \\
\| \\
\text{begin } S_{n1};S_{n2}; \dots ;S_{nk} \text{ end}
\end{array}
$$

Let L_1 and L_2 be two communication closed layers then the distributed programs $\{L_1 + L_2\}$ and $\{L_1 * L_2\}$ are semantically equivalent. A complete proof of the theorem can be found in [3].

Using the CCL safety theorem, we can safely use the SCR during software life cycle development. Since under layer closeness transformation to the DCR preserves the distributed program semantics, the executable distributed program can be the DCR one. Layer closeness must be either verified or imposed. In the application section of this paper, we will show how layer closeness could be imposed using aspect orientation.

The distributed composition rule (DCR) emphasizes processes composition whereas the sequential composition rule emphasizes the composition of logical sub-goals.

4. Semantic Of Aspect Orientation

We use the familiar aspect oriented semantics [11] to provide a common frame or reference that makes it possible to define the structure of the crosscutting concerns inherit in the CCL design.

CCL Join Point: *Joint* points are certain well –defined points in the execution flow of a program. CCL join points are defined at the beginning and the end of each program segment S_{ij}.

CCL Pointcut Designators: *Pointcut designators* identify particular joint points by filtering out a subset of the entire join points in the program flow. CCL pointcut designators are filtering out the joint points at each layer boundaries. Pointcut layer *j* filters out all CCL join points S_{ij} for $i = 1 \dots n$.

CCL Advice: *Advice* declarations are used to define an additional code that runs at join points. CCL advices are used to communicate local process states; or just a relevant subset of it, to establish teamwork cooperation.

CCL Aspect: An aspect is a modular unit of crosscutting implementation. *CCL aspects* are assertions over virtual global states that are verified at runtime. Since a global state is a collection of local states its realization is a crosscutting concern.

Current research has already established the use of aspects in verifying and imposing preconditions and postconditions of design by contract [12]. Aspects make it possible to implement preconditions and postconditions in a modular form. Also a consistent behavior across a large number of operations could be implemented in a much simpler way because of the localization of crosscutting concerns. The contribution of this paper is the extension of the class of properties that can be verified and imposed using aspect orientation approach. The virtual global states as defined in [4] is the distributed programming equivalence to the simple state in sequential programming over which assertions are defined.

To best explain the use of aspect orientation in breaking the tyranny of process composition in distributed programs, we use the well known two-phase commit protocol. A complete formal CCL development of this protocol is given in [9].

The communication closed layers safety theorem applied to this example states that if each of the four layers in closed communications are allowed only between request segments, between vote segments, between decide segments, and between effectuate segments but never between a request segment and a vote segment – then the two compositions are semantically equivalent. This means that we can use the SCR during software life cycle development and the DCR for the actual execution.

These assertions are called virtual global assertions because there might not be any real time at which any one of them holds. Each process reaches its own layer boundaries at a different time.

5. A Formal Aspect Oriented Layer

We can use join points and pointcut designators to define virtual global time and virtual global state.

Let [P_1 || P_2 || … || P_n] be a distributed program **P** composed of n processes P_1 to P_n. And let BEGIN layer-1; layer-2; … ; layer-k END be the complementary program composition into k layers.

Regarding the two-phase commit protocol, the executable program is the one yield by the DCR, so first, we need to wrap each segment with an aspect.

$\forall i \in (1..n), \Pr ocess(i) ::$
BEGIN
    request-*i*;
    vote-*i*;
    decide-*i*;
    effectuate-*i*;
END

There are four CCL join points for every process(i): around request-*i*, around vote-*i*, around decide-*i* and around effectuate-*i*. There are four CCL poitcut designators: $\forall i \in (1..n)$ requiest-*i*, $\forall i \in (1..n)$ vote-*i*, $\forall i \in (1..n)$ decide-*i*, $\forall i \in (1..n)$ effectuate-*i*.

Virtual global times that we would like to express are: the virtual time when the system achieved its *REQUEST* sub-goal, the virtual time when the system achieved its *VOTE* sub-goal, the virtual time when the system achieved its *DECIDE* sub-goal, the virtual time when the system achieved its *EFFECTUATE* sub-goal. At each virtual time we have the associated virtual global state and the virtual global assertions.

The advice code that runs at join points takes a snapshot of the process local state (or just an appropriate subset of all variables that appear in a global invariant) and copies it into a pool of all such snapshots. When a pool is full; all processes have passed their appropriate layer boundaries, the verification of the global assertion can be evaluated. An intelligent decision could be made based on this evaluation. The roles played by states, assertions, and invariants in sequential programming using design by contract discipline can be played by virtual global states, virtual assertions, and virtual invariants in distributed programming. The effectiveness of this approach increases with the degree of logical cooperation and the degree of communication between the processes. Aspect-oriented software development principles support the CCL distributed software development. CCL practical implementation relies on an effective handling crosscutting concerns.

One of the principles in software development is the visibility rule: a significant concern should be syntactically visible. Aspect orientation strength is mainly due to elevating crosscutting concerns to be syntactically visible. CCL strength is mainly due to elevating the cooperative structure of distributed software to be syntactically visible. In the past, we had mostly application where processes, for the most part, did not interfere with each other. Resources management enforced sharing. Now, we see more applications where there is a higher degree of processes cooperation, the processes do not merely share resources, but actually have common goals. Such a distributed program common goals are significant concerns, yet these concerns are not syntactically visible. Given a distributed program, it is impossible to decompose it back to its logical structure in terms of common sub-goals. These types of applications can benefit from an aspect orientation realization of CCL development.

6. Conclusion

Two of the aspect orientation characteristics defined in [2, 5] are illuminating here, the quantification and the implicit invocation. Without these, the implementation of a distributed program using CCL is difficult, rich in code tangling and hence not attractive from practical point. Aspect orientation approach separates the CCL concern from the rest of the program. It enables clean integration between distributed program processes composition and the distributed program layer composition. The tyranny of distributed processes of program composition is gently replaced with co-existence of both process composition and layer composition. Code implementation using this approach composition is not tangle with code implementing using other compositions.

The roles played by states, assertions, and invariants in sequential programming using design by contract discipline. The effectiveness of this approach increases with the degree of logical cooperation and the degree of communication between the processes. CCL practical implementation relies on an effective handling of crosscutting concerns.

7. References

[1] Aspect Oriented Software Development Site http://www.aosd.net

[2] Elrad, T., B. Filman, A. Bader. “Aspect-Oriented Programming,” Communications of ACM, Vol. 44, 10 (2001).

[3] Elrad, T., and N. Frances. “Decomposition of Distributed Programs into Communication Closed Layers,” Science of Computer Programming, North-Holland, 2 (1982).

[4] Elrad, T., and K. Kumar. “State Space Abstraction of Concurrent Systems: A Means to Computation Progressive Scheduling,” Proceedings of the 19th International Conference on Parallel Processing, (1990.)

[5] Filman, R. E., and D. P. Friedman. “Aspect-Oriented Programming is Quantification and Obliviousness, Workshop on Advanced Separation of Concerns,” Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA 2000), Minneapolis, MN, (2000).

[6] Fokkinga, M., M. Poel, and J. Zwiers. “Modular Completeness for Communication Closed Layers,” Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems, Springer-Verlag, (1993).

[7] Gerth, R. T., and L. Shrira. “On Proving Communication Closeness of Distributed Layers,” Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, (1986).

[8] Janssen, W., and J. ZwiersFrom “Sequential Layers to Distributed Processes Deriving a Distributed Mini-mum Weight Spanning Tree Algorithm,” Proceedings of the 11th ACM Symposium on Principles of Distributed Computing, (1992).

[9] Janssen, W., and J. Zwiers. “Protocol Design by Layered decomposition: A Composition Approach,” Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer-Verlag, (1992).

[10] Janssen, W., M. Poel, Q. Xu, and J. Zwiers. “Layering of Real Time Distributed Processes,” Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems, Springer-Verlag, (1994).

[11] Kiczales, G. “Getting Started with Aspect,” Communications of ACM, Vol. 44, No. 10 (2001).

[12] Meyer, B. “Systematic Concurrent Object-Oriented Programming,” Communications of ACM, Vol. 36, 9 (1993).

[13] Manna, Z. and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag, (1991)

[14] Stomp, F. A., and W. P. Roever. “A Correctness Proof of Distributed Minimum Weight Spanning Tree Algorithm,” Proceedings of the 7th International Conference on Distributed Computer Systems, Berlin, West Germany, (1987).

[15] Zwiers, J., and W. Janssen. “Partial Order Based Design of Concurrent Systems,” Proceedings of the REX School/Symposium on A Decade of Concurrency, Noordwijkerhout, (1993).