2-nd Int. Conf. on Dependability of Computer Systems, Szklarska Poreba, Poland, 14-16 June 2007, pp.204-301.
 (DOI 10.1109/DEPCOS-RELCOMEX.2007.53).

Verification of Component Behavioral Compatibility

D.C. Craig and W.M. Zuberek

Department of Computer Science
Memorial University
St.John's, Canada A1B 3X5
{donald,wlodek}@cs.mun.ca

Abstract

In component-based systems, two components are behaviorally compatible if all possible sequences of services requested by one component can be provided by the other component. Verification of this compatibility is essential if subtle software failures, which are difficult to detect and correct, are to be eliminated. For verification of compatibility, the behavior of interacting components, at their interfaces, is modeled by labeled Petri nets with labels representing the requested and provided services. The paper discusses the verification process for several classes of interface languages, with emphasis on the algorithmic aspects of verification.

1 Introduction

The difficulties and challenges involved in the development of large–scale software systems are fairly well documented [4, 16] and several strategies have been proposed to address these difficulties [16, 17]. Concepts related to software architecture [1, 6, 9, 14] are the most promising attempt to provide the basis of a new set of techniques for the next generation of software-intensive solutions [2, 3]. Software architecture uses components as the basic building blocks of software systems.

Components can be considered as the basic functional units and the fundamental data types in architectural design. Components represent high-level software models; they must be generic enough to work in a variety of contexts and in cooperation with other components, but they also must be specific enough to provide easy reuse [18, 19]. Although there are a variety of component definitions, they seem to be consistent with respect to the most characteristic properties of components:

- components are units of independent deployment,
- components are units of third-party composition,
- components have no (externally) observable state.

These properties have several implications. For a component to be independently deployable, it needs to be well separated from its environment and other components. A component, therefore, encapsulates its constituent features. Also, as a unit of deployment, a component will never be deployed partially [19].

For a component to be composable with other (third-party) components, it needs to be sufficiently self-contained. Also, it needs to come with clear specifications of what it requires and what it provides. In other words, a component needs to encapsulate its implementation and interact with its environment by means of well-defined interfaces.

Finally, a component would not have any (externally) observable state – it is required that the component cannot be distinguished from copies of itself. Possible exceptions to this rule are attributes contributing to the component's functionality, such as serial numbers used for accounting. The specific exclusion of observable state allows for permissible technical uses of state that can be crucial for performance without affecting the observable behavior of a component [18].

Due to the stateless nature of components, it makes little sense to have multiple copies of the same component in the same operating system as these would be mutually indistinguishable anyway. In many current approaches, components are heavyweight units with exactly one instance in a system. For example, a database server could be a component [19].

The interface of a component defines the component's access points [18]. These points allow clients of a component to access the services provided by the component. Normally, a component will have multiple interfaces corresponding to different access points. Each access point may provide a different service, catering to different client needs.

Two interacting components are compatible if all services that are requested by one component are provided by the other components. Such "static" compatibility can usually be checked quite easily, but it does not prevent more subtle errors which are due to some limitations on the ordering of services. Therefore a "dynamic" (or behavioral) compatibility is used, and two components are compatible in the behavioral sense if all possible sequences of services requested by one of the interacting components can be provided by the other component. This paper discusses algorithmic aspects of the verification of behavioral compatibility of components which are represented by labeled Petri nets; it is a continuation of an earlier paper [8] that discussed behavioral compatibility in a conceptual way (with numerous illustrations in the form of Petri net models).

Section 2 recalls Petri net models of the behavior of components. The concept of compatibility and its verification is addressed in Section 3, while Section 4 contains concluding remarks with some directions for future work.

2 Component models

The behavior of a component, at its interface, can be represented by a labeled Petri net [7]:

$$\mathcal{M}_i \;=\; (P_i, T_i, A_i, S_i, m_i, \ell_i, F_i),$$

where P_i and T_i are disjoint sets of places and transitions, respectively, A_i is the set of directed arcs connecting places with transitions and transitions with places, $A_i \subseteq P_i \times T_i \cup T_i \times P_i$, S_i is an alphabet representing the set of services that are associated with transitions by the labeling function $\ell_i : T_i \to S_i \cup \{\varepsilon\}$ (ε is the "empty" service; it labels transitions which do not represent services), m_i is the initial marking function $m_i : P_i \to \{0, 1, ...\}$, and F_i is the set of final markings (which are used to indicate the end of sequences of firings).

In order to represent component interactions, the interfaces are divided into *provider* interfaces (or p-interfaces) and *requester* interfaces (or r-interfaces). In the context of a provider interface, a labeled transition can be thought of as a service provided by that component; in the context of a requester interface, a labeled transition is a request for a corresponding service. It is required that in each p-interface there is exactly one labeled transition for each provided service:

$$\forall t_i, t_j \in T : \ell(t_i) = \ell(t_j) \Rightarrow t_i = t_j.$$

The label assigned to a transition represents a service or some unit of behavior. For example, the label can represent a conventional procedure or method invocation. It is assumed that if the p-interface requires parameters from the r-interface, then the appropriate number and types of parameters are delivered by the r-interface. Similarly, it is assumed that the p-interface provides an appropriate return value, if such a value is required. The equality of symbols representing component services (provided and requested) implies that all such requirements are satisfied.

Component behavior is determined by the set of all possible sequences of services (required or provided by a component) at a particular interface. Such a set of sequences is called the interface language.

Let $\mathcal{F}(\mathcal{M})$ denote the set of firing sequences in $\mathcal{M}$. The interface language of a component represented by a labeled Petri net $\mathcal{M}$, $\mathcal{L}(\mathcal{M})$, is the set of all labeled firing sequences of $\mathcal{M}$:

$$\mathcal{L}(\mathcal{M}) \;=\; \{\ell(\sigma) \mid \sigma \in \mathcal{F}(\mathcal{M})\},$$

where $\ell(t_{i_1} t_{i_2} ... t_{i_k}) = \ell(t_{i_1})\ell(t_{i_2})...\ell(t_{i_k})$.

3 Component compatibility

Interface languages of interacting components can be used to define the compatibility of components; a requester component $\mathcal{M}_i$ is compatible with a provider component $\mathcal{M}_j$ if and only if all sequences of services requested by $\mathcal{M}_i$ can be provided by $\mathcal{M}_j$, i.e., if and only if:

$$\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j).$$

3.1 Compatibility verification

Verification of component compatibility depends upon the type of interface languages. If these languages are regular (in the sense of Chomsky hierarchy [11]), verification can be performed directly on the languages. If, however, the languages are non-regular, an indirect approach is needed in which the components are first composed, and then the behavior of the composition is analyzed. The composition of component models can be performed in different ways, resulting in composed models with different properties.

3.2 Regular compatibility

If the languages of interacting requester and provider components are regular, checking the compatibility is relatively straightforward because the compatibility relation can be expressed as:

$$\mathcal{L}(\mathcal{M}_i) \cap \overline{\mathcal{L}(\mathcal{M}_j)} \;=\; \emptyset$$

where $\emptyset$ denotes the empty set, and $\overline{\mathbf{A}}$ is the complement of the set $\mathbf{A}$. Since the class of regular languages is closed under the operations of complementation and set intersection, the compatibility can be verified by performing the corresponding operations on finite automata representing the requester and provider languages. Let $\mathcal{A}_i = (\mathbf{\Sigma}_i, \mathbf{S}, \delta_i, s_i, \mathbf{F}_i)$ be a deterministic finite automaton defining $\mathcal{L}(\mathcal{M}_i)$, and let $\mathcal{A}_j = (\mathbf{\Sigma}_j, \mathbf{S}, \delta_j, s_j, \mathbf{F}_j)$ be a deterministic finite automaton defining $\mathcal{L}(\mathcal{M}_j)$. Both automata can be easily derived from the net models of the components behavior [7]. A product of deterministic finite automata

is a well–known operation [11], however, since the interface automata can be quite large, the product operation should be performed in a way that eliminates all inessential pairs of states (i.e., pairs of states which cannot be reached from the initial state). This can easily be done, as outlined in the following algorithm which also checks if a final state can be reached in the product automaton (i.e., if the intersection of languages is empty) and sets the variable *Empty* correspondingly (for an ordered pair $s = (a, b)$, the first element, a, is extracted by the operation *first(s)*, and the second element, b, by *second(s)*; also, the function *append(x,a)* returns a sequence x extended by an element a, *head(x)* returns the initial element of x, and *tail(x)* returns the sequence x without its initial element; sequences are denoted by angle brackets "$\langle$" and "$\rangle$"):

$Empty := \text{YES};$
$States := \{(s_i, s_j)\};$
$New := \langle (s_i, s_j) \rangle;$
while $New \neq \langle\rangle$ **do**
  $s := head(New);$
  $New := tail(New);$
  for each a **in** S **do**
    $s' := (\delta_i(first(s), a), \delta_j(second(s), a));$
    if $s' \notin States$ **then**
      $States := States \cup \{s'\};$
      $New := append(New, s');$
      if $s' \in \mathbf{F}_i \times (\mathbf{\Sigma}_j - \mathbf{F}_j)$ **then** $Empty :=$NO **fi**
    fi
  od
od

If the variable *Empty* remains set to YES after the termination of this algorithm, the compatibility relation is satisfied, and all sequences of services requested by $\mathcal{M}_i$ are provided by $\mathcal{M}_j$.

3.3 Non-regular compatibility

It is known that Petri net languages include all regular languages as well as some context-free languages and even context-sensitive languages [13]. For non-regular languages the approach outlined earlier cannot be used because the class of context-free languages is not closed under complementation, so a different approach is needed for verification of compatibility of components with non-regular behavior.

The compatibility of interacting components can be verified by composing the component models into one model and checking the properties of this model. The composition, however, can be performed in several ways, resulting in models with different properties.

The COSY–style composition [12] uses the fusion of transitions labeled by the same services (with some additional elements to distinguish repeated requests of the same service). The consequence of such an approach is that the composition corresponds to the intersection of languages of the provider and requester interfaces:

$$\mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j).$$

The verification of the compatibility is thus checking the equality:

$$\mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j) \;=\; \mathcal{L}(\mathcal{M}_i)$$

which is as difficult as the verification of the original compatibility relation.

The idea behind the CORD (compatible or deadlocked) composition [7] is to make the language of composed interfaces equal to the language of the requester, or to introduce a deadlock when the requested sequence of services cannot be provided by the other component. The verification of the component compatibility is thus equivalent to deadlock detection in the composed model. The details are given in [7].

There are two basic methods of deadlock detection in Petri nets. If the net is bounded, reachability analysis can be used for exhaustive exploration of the marking space, with deadlock detection as a straightforward addition to the exploration procedure (in the following algorithm, the function *enable(m)* returns the set of transitions enabled by the marking m, and the function *fire(m,t)* returns the marking that is obtained from m by firing transition t):

$Deadlock := \emptyset$;
$Markings := \{m_0\}$;
$New := \langle m_0 \rangle$;
while $New \neq \langle\rangle$ **do**
  $m := head(New)$;
  $New := tail(New)$;
  $E := enable(m)$;
  if $E = \emptyset$ **then**
    $Deadlock := Deadlock \cup \{m\}$
  else
    for each t **in** E **do**
      $m' := fire(m, t)$;
      if $m' \notin Markings$ **then**
        $Markings := Markings \cup \{m'\}$;
        $New := append(New, m')$;
      fi
    od
  fi
od

If the set variable *Deadlock* remains empty after termination of the algorithm, the marked net $\mathcal{M} = (\mathcal{N}, m_0)$ is free from deadlocks, otherwise *Deadlock* contains all dead markings which are reachable from m_0.

If the composed model is unbounded, or if the space of reachable markings is finite but unreasonably large, the structural approach can be used.

The structural approach to deadlock detection is based on siphons [10, 13] (in earlier publications siphons were called deadlocks) which are defined as such subsets of places $P_i \subseteq P$, for which:

$$Inp(P_i) \subseteq Out(P_i),$$

where $Inp(P_i)$ and $Out(P_i)$ are the input and output sets of P_i:

$$Inp(P_i) = \bigcup_{p \in P_i} \{t \in T \mid (t, p) \in A\}, \quad Out(P_i) = \bigcup_{p \in P_i} \{t \in T \mid (p, t) \in A\}.$$

The characteristic property of siphons is that once a siphon P_i becomes unmarked under a marking m (i.e., m assigns zero tokens to all places of P_i), P_i remains unmarked for all markings reachable from m.

It can be shown [5] that in a deadlocked net, all unmarked places constitute a siphon. The siphon-based approach to deadlock detection systematically checks if the net contains a proper siphon (a siphon is proper if its input set is a subset of its output set) that can become unmarked by some firing sequence, and if such a siphon is identified, the initial marking is modified by the firing sequence, and the check continues for the remaining (marked, proper) siphons until a deadlock is identified, or until no further progress can be done. Linear programming is used to find the firing sequence that minimizes the number of tokens in each proper siphon of the analyzed net (if such a sequence exists) [5, 15].

The drawback of this approach is that the number of siphons in net models of interfaces is often quite large, so, for practical applications, a significant reduction of this number is needed.

A minimal siphon is usually defined as a siphon which does not contain any other siphon. Usually, net models contain just a few minimal siphons. However, minimal siphons rarely determine the deadlocks. Therefore another type of siphons is needed which is known as basis siphons. Basis siphons are siphons from which all other siphons can be obtained by the union operation. Minimal siphons are basis siphons, but usually some basis siphons are not minimal. The number of basis siphons is typically significantly larger than the number of minimal siphons.

The proposed deadlock detection procedure [7] is a two stage one; first the (hopefully small) set of minimal siphons is used for checking the deadlock, and when no further progress can be made using minimal siphons, a switch is made to use basis siphons (since each deadlock corresponds to one or a union of several unmarked basis siphons). Let $\mathcal{S}_M$ be the set of minimal siphons of a marked net $\mathcal{M}$, and let $\mathcal{S}_B$ be the set of its basis siphons; deadlock detection is performed by the invocation "$deadlock(m_0, \mathcal{S}_M, \mathcal{S}_B)$" where the recursive boolean function *deadlock* is:

```
function deadlock(m, X, Y) : boolean;
begin
    if enable(m) = ∅ then return TRUE fi;
    if X ≠ ∅ then
        for each x in X do
            (v, n) := LPminimize(x, m);
            if nonzero(v) ∧ n = 0 ∧ feasible(v, m) then
                m' := m + C × v;
                X' := marked(X, m');
                if deadlock(m', X', Y) then return TRUE fi
            fi
        do
    fi;
    if Y ≠ ∅ then return deadlock(m, Y, ∅)fi;
    return FALSE
end
```

The function *enable(m)*, as before, returns the set of transition enabled by m. *LPminimize* is the linear programming procedure which tries to minimize the number of tokens assigned to the siphon x by the marking m by finding an appropriate firing sequence; it returns a firing vector v (indicating, for each transition, the number of times it occurs in the firing sequence) and n, the final number of tokens in the siphon x. If v is a nonzero vector and $n = 0$ and the firing vector v is feasible at m (i.e., there exists a firing sequence that begins at m and corresponds to v), then the current siphon becomes unmarked, and the checking continues for a reduced set of siphons and a modified marking. $\mathbf{C}$ is the incidence

(or connectivity) matrix of the analyzed net, and the function *marked(X,m)* returns the set of all those siphons in X which are marked by m. The last section of *deadlock* performs the switch from minimal siphons to basis siphons (and continues deadlock detection).

If the function *deadlock* returns TRUE, the analyzed net contains a deadlock (a simple modification of the function can provide a firing sequence creating this deadlock); if the function returns FALSE, the net is deadlock–free.

The linear programming procedure returns a firing vector minimizing the number of tokens in the analyzed siphon. Such a firing vector may have no implementation in the form of a firing sequence, i.e., it may be infeasible for a given marking m. Therefore the feasibility of firing vectors is checked by another recursive (boolean) function:

```
function feasible (v,m) : boolean;
begin
    if zero(v) then return TRUE fi;
    for each t in enable(m) do
        if v[t] > 0 then
            v' := v;
            v'[t] := v'[t] − 1;
            m' := fire(m, t);
            if feasible(v', m') then return TRUE fi
        fi
    od;
    return FALSE
end
```

If the function returns TRUE, the firing vector v is feasible for marking m; if the returned value is FALSE, such a firing sequence does not exist, and v is infeasible for m.

4 Concluding Remarks

The strategy described in this paper is an initial, but important step in the continuing evolution of the design and construction of dependable software systems. Establishing a well defined and formal method for determining the extent to which two or more components are able to reliably interact can serve to significantly enhance reuse of software components in a given software architecture. Ultimately, this may contribute to the successful evolution of a deployed component-based software system.

Although the discussion focused on interactions of two components, more general scenarios can be discussed in a similar way; a number of results for systems with multiple requester and multiple provider components can be found in [7].

The proposed approach can be extended in many ways, for example, temporal characteristics of components can be included into net models and used for performance analysis [20]. Also, for systems composed of many requester and provider components, an incremental approach would be interesting as it would allow a software architect to "reuse" the verification steps for subsystems that are not affected by modifications.

An interesting related problem is how to obtain Petri net models of components; would it be practical to generate such models from component specifications or, perhaps, from the implementation code?

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada through Grant RGPIN-8222.

References

[1] S.T. Albin, "The art of software architecture: design methods and techniques"; Wiley 2003.

[2] A.W. Brown, "Large–scale component–based development"; Prentice–Hall 2000.

[3] A.W. Brown, "An overview of components and component–based development"; in "Advances in Computers", vol.54 – Trends in Software Engineering, pp.1-34, Academic Press 2001.

[4] D. Bugden, "Software design" (2 ed.); Addison-Wesley 2003.

[5] F. Chu, X. Xie, "Deadlock analysis of Petri nets using siphons and mathematical programming"; *IEEE Trans. on Robotics and Automation*, vol.13, no.6, pp.793-804, 1997.

[6] P. Clements, R. Kazman, M. Klein, "Evaluating software architectures - methods and case studies"; Addison-Wesley 2002.

[7] D.C. Craig, "Compatibility of software components – modeling and verification"; Ph.D. Thesis, Department of Computer Science, Memorial University, St.John's, Canada A1B 3X5, 2006.

[8] D.C. Craig, W.M. Zuberek, "Compatibility of software components – modeling and verification"; Proc. Int. Conf. on Dependability of Computer Systems, Szklarska Poreba, Poland, pp.11-18, 2006.

[9] D. Garlan, D.E. Perry, "Introduction to the special issue on software architecture"; *IEEE Trans. on Software Engineering*, vol.21, no.4, pp. 269-274, 1995.

[10] M. Hack, "Analysis of production schemata by Petri nets"; Technical Report TR-94, Massachusetts Institute of Technology, Cambridge, MA, USA, 1972.

[11] J.E. Hopcroft, R. Motwani, J.D. Ullman, "Introduction to automata theory, languages, and computations" (2 ed.); Addison Wesley 2001.

[12] R. Janicki, P.E. Lauer, "Specification and analysis of concurrent systems – the COSY approach"; Springer-Verlag 1992.

[13] T. Murata, "Petri nets: properties, analysis, and applications", *Proceedings of the IEEE*, vol.77, no.4, pp.541-580, 1989.

[14] M. Shaw, D. Garlan, "Software architecture: perspectives on an emerging discipline"; Prentice Hall 1996.

[15] M. Silva, E. Teruel, J. Couvreur, "Linear algebra in and linear programming techniques for the analysis of place/transition net systems"; in "Lecture on Petri nets - basic models" (Lecture Notes in Computer Science 1491), pp.309-373, Springer-Verlag 1998.

[16] I. Sommerville, *Software Engineering* (6 ed.), Addison-Wesley 2001.

[17] D.H. Steinberg, D.W. Palmer, "Extreme software engineering: a hands-on approach", Pearson/Prentice Hall 2004.

[18] C. Szyperski (with D. Gruntz, S. Murer), "Component software: beyond object–oriented programming" (2 ed.); Addison-Wesley 2002.

[19] C. Szyperski, "Component software and the way ahead"; in "Foundations of component–based systems", G.T. Leavens, M. Sitaraman (eds.), pp.1-20, Cambridge University Press 2000.

[20] W.M Zuberek, "Timed Petri nets – definitions, properties and applications"; *Microelectronics and Reliability* (Special Issue on Petri Nets and Related Graph Models), vol.31, no.4, pp.627-644, 1991.