Compositional Checking of Communication among Observers*

Ralf Pinger and Hans-Dieter Ehrich

Abteilung Informationssysteme, Technische Universität Braunschweig
Postfach 3329, D-38023 Braunschweig, Germany
{R.Pinger|HD.Ehrich}@tu-bs.de
Fax: +49-531-3913298

Abstract. Observers are objects inside or outside a concurrent object system carrying checking conditions about objects in the system (possibly including itself). In a companion paper [EP00], we show how to split and localise checking conditions over the objects involved so that the local conditions can be checked separately, for instance using model checking. As a byproduct of this translation, the necessary communication requirements are generated, taking the form of RPC-like action calls (like in a CORBA environment) among newly introduced communication symbols. In this paper, we give an algorithmic method that matches these communication requirements with the communication pattern created during system specification and development. As a result, correctness of the latter can be proved. In case of failure, the algorithm gives warnings helping to correct the communication specification.

Keywords. compositionality, distributed logic, model checking, modelling and design, object system, temporal logic, verification.

1 Introduction

In this paper, we elaborate on a novel compositional checking technique for distributed object systems that may be used for deductive verification, model checking or testing. In a companion paper [EP00], we show how to split and localise checking conditions over the objects involved so that the local conditions can be checked separately. Here we show how the necessary communication and thus the overall object system can be checked.

An object system in our sense is a community of sequential objects operating concurrently and communicating via synchronous RPC-like message passing (like in a CORBA environment). Objects may be distributed over sites.

The complexity of system verification is exponential in the number of concurrent components: the system state space "explodes" in size. To overcome this problem, compositional specification and verification techniques have been widely discussed. The idea is best expressed in [dR97]:

* This work was partially supported by the EU under ESPRIT IV Working Group 22704 ASPIRE.

H. Hussmann (Ed.): FASE 2001, LNCS 2029, pp. 32–44, 2001.

> "The purpose of a compositional verification approach is to shift the burden of verification from the global level to the local, component, level, so that global properties are established by composing together independently (specified and) verified component properties".

The benefit of compositional verification is obvious: instead an exponential increase of the overall verification amount, compositional verification has a linear complexity with respect to the number of components.

The first compositional verification approach was given by [MC81] where a rule for composing networks was given that is analogous to pre- and postconditions in sequential program proving. In [Pnu85], Pnueli defined the *assume-guarantee* approach that is very powerful but requires some human interaction for dividing the global property into suitable modular properties. Abadi and Lamport [AL93,AL95] used a modular way of specifying systems and were able to include liveness properties in the guarantee parts of the assume-guarantee rules. Moreover, their approach deals with fairness and hiding.

Clarke, Long and McMillan adapted these techniques for model checking [CLM89]. Their approach employs interfaces which represent only that part of a subsystem which is observable by one particular other subsystem. By composing such an interface with the corresponding subsystem, formulae can be verified in the smaller system which are true in the entire system. The performance of this approach depends heavily on the size of the interfaces, so it is best suited for loosely coupled systems. Another limitation deals with the fact that the interface rule can only handle boolean combinations of temporal properties of the individual processes.

Grumberg and Long as well as Josko developed machine supported versions of compositional verification. Grumberg and Long used a subclass of CTL which does not allow existentially quantified path formulae [GL94]. Another limitation is that all components of the system have to synchronise on a transition. This is suitable for clocked systems like, e.g., in hardware but it is not appropriate for software components which are distributed over a computer network or even over the internet. Josko [Jos89] used a modular but quite restricted version of CTL for writing assertions in an assume-guarantee way. Moreover, the number of verification steps grows exponentially with the number of until/unless formulae in the assumptions.

Fiadeiro and Maibaum give foundational aspects of object-oriented system verification in [FM95]. They have developed sound inference rules for reasoning about object system. However, there is no tool support for this approach.

The idea we put forward here does not restrict the logic: we employ a distributed extension of full CTL for specifying the checking conditions. The added "distributed" facility consists of nesting statements about communication with concurrent objects. This logic, called D_1, can be automatically translated to the logic D_0 that also employs full CTL locally but has a very restricted communication mechanism reflecting RPC-like action calling.

Our method is orthogonal to the other compositionality approaches, it may be combined with the others. Which combinations are useful is subject to further

study. In a sense, traditional compositionality approaches can be embedded into our approach: the traditional global view may be seen as bound to an "external observer" in our sense.

Object-oriented modelling and design is widely accepted; this is given evidence, among others, by the fact that UML has become an industrial standard. The work reported here is performed in the context of developing the TROLL language, method and tools for information systems development [HDK+97,DH97, GKK+98] where we put some effort in investigating the logic and mathematical foundations of the approach. The semantics of TROLL is based on distributed temporal logic like the one used in this paper. Its use for specification is described in [ECSD98,EC00].

In the next section, we give a brief and informal introduction to our approach, details can be found in [EP00]. We use the example given there in order to demonstrate how to distribute "global" assertions over the objects involved, introducing communication requirements in the form of RPC-like communication rules. Here we capitalise on a result on translating distributed logics published in [ECSD98,EC00]. In the 3rd section we present the main result of this paper, an algorithmic method that matches these communication requirements with the communication pattern created during system specification and development.

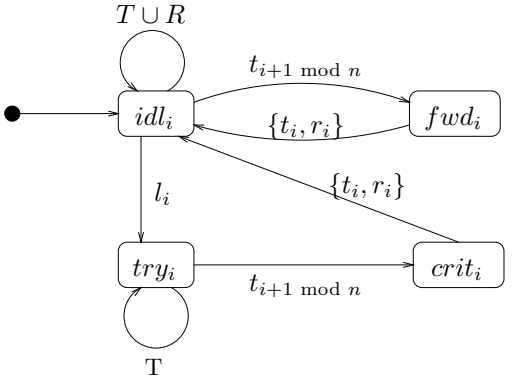
2 Distributing Checking Conditions

The idea is illustrated by an n-process mutual exclusion example where we identify an object with a process. A process may be idle or trying to enter the critical region; only one process at a time is allowed to enter its critical region.

Example 1 (Mutual Exclusion). We present a "self-organising" version of the n-process mutual exclusion problem that is advantageous in cases where the system is at rest most of the time because then there is no communication traffic. In the beginning, all processes $P_0, \ldots, P_{n-1}$ are idle, and the system is at rest. When k processes try to enter the critical region, $0 < k \leq n$, one is chosen nondeterministically and a token ring of all processes is initialised, giving all processes in turn a chance to enter the critical region if they want to. Whenever all processes are idle, the system may go at rest again, i.e., the token ring is put dormant.

A state diagram of P_i is shown in figure 1. Each process $P_i, i \in \{0, \ldots, n-1\}$ has four states: $\{idl_i, try_i, crit_i, fwd_i\}$. Process P_i has the token iff it is in one of its right-hand side states, $crit_i$ or fwd_i, respectively. A transition carrying a set of actions as a label is an abbreviation for a set of transitions between the same pair of states, one for each action in the set.

The token ring is established as follows. Each process P_i receives the token from its right-hand neighbour $P_{i+1 \bmod n}$ and sends it to its left-hand neighbour $P_{i-1 \bmod n}$. Transmitting the token is modelled by two actions for each process P_i: $t_{i+1 \bmod n}$ represents receiving the token, and t_i represents forwarding it. Of course, forwarding in P_i synchronises with receiving in $P_{i-1 \bmod n}$, represented



$$T = \{t_j \mid j \neq i + 1 \bmod n\}, R = \{r_k \mid k \neq i\}$$

Fig. 1. State diagram of process i

by sharing the same action t_i. The communication between these processes is easily expressed by action calling formulae: $P_k.(t_i \rightarrow P_j.t_i)$ with $k \neq j$.

On receiving the token, a process enters the critical region if it wants to, i.e., if it is in its *try* state. Otherwise, i.e., if it is in its *idl* state, it simply forwards the token, moving to the *fwd* state when receiving the token and back to the *idl* state when forwarding it. The token is also forwarded when the process leaves the critical region, going into its *idl* state. The process may move from its *idl* state to its *try* state via action l_i any time without synchronising with other processes, indicating its spontaneous desire to enter the critical region.

When the system is at rest, one or more processes may spontaneously enter their *try* states, and one of these will proceed into the critical region, making sure by direct communication with all the others that they keep away. At the same time, the token ring is put into operation. Whenever a process terminates its critical region or forwarding state and all others are idle, the system may go at rest again, i.e., all processes are idle and the token ring is dormant. This is the case when the last active process P_i chooses its reset transitions r_i, synchronising with the r_i reset transitions of all other processes. The latter are only applicable in their *idl* states, ensuring that all processes are in their *idl* states after the move.

For the sake of simplicity, each process synchronises with all the others when entering the critical region, also when the token ring is active when this is not really necessary. In our solution shown in figure 1, this is accomplished by the loops at the *idl* and *try* states, synchronising with token actions of all the other processes. Also, we abstain from enforcing that the system goes at rest again as soon as it can, the token ring may nondeterministically remain active for a while (or even forever).

We concentrate on checking safety.

$$P_i.\,\mathsf{A}\,\mathsf{G}(crit_i \;\Rightarrow \bigwedge_{j=1, j\neq i}^{n} P_j.\neg\, crit_j) \;\text{ for all }\; i \in \{0, \ldots, n-1\}$$

The assertions are written in the distributed version of CTL that we introduce below. It should be intuitively clear what they mean. The i-th safety condition makes an assertion about all the other processes: they must not be in the critical region when the i-th process is.

In order to check safety, and assuming that the communication is correct, all we have to do is to prove the following.

$$P_i.(crit_i \;\Rightarrow\; t_{i+1 \bmod n}) \quad \text{for all } i \in \{0,\dots,n-1\}, \text{ and}$$
$$P_j.(t_{i+1 \bmod n} \;\Rightarrow\; \neg\, crit_j) \text{ for all } j \neq i,\ j \in \{0,\dots,n-1\}$$

The first condition says that P_i cannot be in the critical region unless it has been entered via action $t_{i+1 \bmod n}$. The second condition says that if $t_{i+1 \bmod n}$ has happened, no other process can be in the critical region. These conditions are local, they are obviously satisfied by our solution.

We note in passing that also other correctness criteria (cf. [HR00]) are satisfied by our solution: *liveness* (each process which is in its try state will eventually enter the critical section), *non-blocking* (a process can always request to enter the critical region) and *no-strict-sequencing* (processes need not enter the critical region in strict sequence). We do not elaborate on these conditions here. A compositional proof for the liveness property can be found in [EP00].

It remains to prove that communication indeed works correctly. What must be shown is the following.

$$P_i.(a_i \;\Rightarrow\; P_j.a_i) \text{ and } P_j.(a_i \;\Rightarrow\; P_i.a_i) \text{ for } \; i,j \in \{0,\dots,n-1\}.$$

where a may stand for t or r: each token or reset action synchronises with its counterparts in the other processes. For $i = j$, the formulae are trivial. The precise meaning of these formulae will be made clear in the next section.

Observers are objects inside or outside a concurrent object system carrying checking conditions about objects in the system (possibly including itself). Let I be a finite set of observers (or, rather, observer identifiers). Each observer $i \in I$ has an individual set A_i of action symbols expressing which action happens during a given transition, and an individual set Σ_i of state predicate symbols expressing which attributes have which values in a given state. Let $\Sigma = (I, \{A_i, \Sigma_i\}_{i\in I})$ be the observer signature. Our multi-observer logic is like the distributed logic D_1 described in [ECSD98,EC00] but instantiated with CTL, the computation tree logic due to E. Clarke and A.E. Emerson [CE81]. This logic $\mathsf{D}_1(\Sigma)$ — or D_1 for short when Σ is clear from context — is defined as follows.

Definition 1. The *observer logic* D_1 is the I-indexed family of local logics $\{\mathsf{D}_1^i\}_{i\in I}$ where, for each $i \in I$, D_1^i is CTL over A_i and Σ_i with the added possibility of using formulae in $\mathsf{D}_1^j, j \in I, j \neq i$ as subformulae.

A given D_1 formula $i.(\dots j.\psi \dots)$ with a communication subformula $j.\psi \in \mathsf{D}_1^j, j \neq i$, means that i communicates — or synchronises — with j and asserts that ψ is true for j at this moment of synchronisation. Thus, each D_1 formula

is bound to an observer, and each formula of another observer may serve as a subformula, representing communication with that observer.

For instance, the safety formulae in example 1 are in D_1: the subformulae $P_j.\neg\, crit_j, j \in \{1,\ldots,n\}, j \neq i$, are communication subformulae.

As shown in example 1 above, these assertions are transformed to purely local formulae and primitive communication formulae,

$$P_i.(a_i \;\Rightarrow\; P_j.a_i) \text{ and } P_j.(a_i \;\Rightarrow\; P_i.a_i) \text{ for } \; i,j \in \{0,\ldots,n-1\}.$$

All these formulae are in D_1. The latter communication subformulae characterise RPC, the basic middleware communication mechanism.

Formulae of this kind constitute the sublogic D_0 which we define below.

First we give a more detailed and more precise definition of D_1 semantics. We assume a family $\mathcal{M} = \{\mathcal{M}_i\}_{i\in I}$ of models to be given, one for each observer, where $\mathcal{M}_i = (S_i, \rightarrow_i, L_i), i \in I$. For each observer i, S_i is its set of states, $\rightarrow_i \subseteq S_i \times A_i \times S_i$ is its action-labelled state transition relation, and $L_i : S_i \rightarrow_i 2^{\Sigma_i}$ is its state labelling function. We write $s \xrightarrow{a}_i t$ for $(s,a,t) \in \rightarrow_i$. As in example 1, we may extend the model to allow for finite sets B of actions as transition labels: $s \xrightarrow{B} s' \;\Leftrightarrow\; \forall a \in B : s \xrightarrow{a} s'$.

We assume that the reader is familiar with the conventional CTL semantics (cf., e.g., [HR00]): it defines the meaning of $\mathcal{M}_i, s_i \models_i \varphi$ for every observer i where $s_i \in S_i$ and $\varphi \in \mathsf{D}_1^i$, as long as φ does not contain any communication subformula — and as long as there are no action symbols, i.e. $|A_i| = 1$ for every observer i.

Action symbols are easily introduced by "pushing them into the states": given a model $\mathcal{M} = (S, \rightarrow, L)$ where $\rightarrow \subseteq S \times A \times S$ and $L : S \rightarrow 2^{\Sigma}$, we define the model $\mathcal{M}' = (S', \rightarrow', L')$, $\rightarrow' \subseteq S' \times S', L' : S' \rightarrow A_i \times 2^{\Sigma_i}$, by

$$\begin{aligned} S' &= S \times A \times S \\ (s_1 \xrightarrow{a_1} s_2) \rightarrow' (s_3 \xrightarrow{a_2} s_4) &\text{ iff } \; s_2 = s_3 \\ L'(s_1 \xrightarrow{a} s_2) &= (a, L(s_2)) \end{aligned}$$

for all $s_1, s_2, s_3, s_4 \in S'$ and all $a, a_1, a_2 \in A$. Of course, $s \rightarrow' t$ stands for $(s,t) \in \rightarrow'$. We use action symbols a also as action occurrence predicates meaning that action a occurs during a transition (in $\mathcal{M}$) or during a state (in $\mathcal{M}'$), respectively.

Our construction is different from the ACTL construction given in [dNV90] where a new state s_a is introduced on each transition labelled a. Our construction gives us states with action symbols as state predicates; this is essential for the translation of D_1 to D_0 as described in the following section.

Capturing communication is not quite so easy. Note that with treating the actions as described above, we now synchronise states rather than transitions. However, whether a state of one observer synchronises with a given state of another one does not only depend on the current state of the former but on what happened before in the latter. We refer to [EC00,EP00] for more details on this matter.

Now we demonstrate how D_1 formulae can be translated to D_0 formulae in a sound and complete way where $\mathsf{D}_0 \subseteq \mathsf{D}_1$ is a sublogic featuring sketched above. Full details of the translation as well as soundness and completeness proofs are given in [EC00]. We first introduce D_0, outline the translation rules, and show how the translation can be utilised to transform global assertions (in D_1) into multiple local assertions (in D_0). This way, the communication primitives and the local assertions for the objects involved are separated so that they together guarantee the original global assertion.

For defining D_0, we assume again that a family of object signatures $\Sigma = (I, \{A_i, \Sigma_i\}_{i \in I})$ is given which we do not show explicitly in our notation.

Definition 2. The *object sublogic* $\mathsf{D}_0 \subseteq \mathsf{D}_1$ is the I-indexed family of local logics $\{\mathsf{D}_0^i\}_{i \in I}$ where, for each $i \in I$, each formula in D_0^i is either a CTL formula over A_i, Σ_i and $@j, j \in I$, or a communication formula of the form $a_i \Rightarrow j.a_j, j \in I, j \neq i, a_i \in A_i, a_j \in A_j$.

The predicates $i.@j$ mean the same as $i.j.\neg\bot$ in D_1, namely that i communicates with j. But this must be given an extra symbol in D_0 because $i.j.\neg\bot$ is not a formula in D_0. Note that $i.@i$ is trivially true: each object synchronises with itself all the time.

Examples of D_0 formulae have been given above. For the straightforward proof that D_0 is indeed a sublogic of D_1, we refer to [EC00].

The idea of the translation is as follows. Let an assertion $\varphi \in \mathsf{D}_1^i$ be given saying, somewhere in context, that an assertion ψ holds for j,

$$\varphi \;\Leftrightarrow\; i.(\ldots j.\psi \ldots),$$

where $j.\psi$ is in D_0, i.e., ψ does not have a communication subformula. The meaning is that i transmits a "message" q to j,

$$\overline{\varphi} \;\Leftrightarrow i.(\ldots q \ldots),$$

that is equivalent in j to the validity of ψ during communication with i,

$$\delta \;\Leftrightarrow j.(q \Leftrightarrow @i \wedge \psi)$$

q is a new action symbol in i as well as in j, i.e., there are local actions $i.q$ in i and $j.q$ in j "calling" each other. This communication is captured by

$$\alpha \;\Leftrightarrow i.(q \Rightarrow j.q)$$
$$\beta \;\Leftrightarrow j.(q \Rightarrow i.q)$$

The resulting formulae δ, α and β are in D_0. $\overline{\varphi}$ may still be in $\mathsf{D}_1 - \mathsf{D}_0$, but the number of communication subformulae has decreased by one. With α and β as defined above, the translation step is sound (φ implies $\overline{\varphi}$ and δ) and complete ($\overline{\varphi}$ and δ imply φ).

The translation of arbitrary D_1 formulae is accomplished by iterating the above step inside–out until no communication subformulae are left. Together

with the corresponding RPC-like communication formulae like α and β, the communication infrastructure is established for reducing global to local model checking. Clearly, the translation process terminates. Its soundness and completeness is proven in [EC00].

We illustrate the idea by the mutual exclusion example (example 1).

Example 2 (Mutual Exclusion cont.). In order to keep the example small but not too trivial, we take $n = 3$. The table in figure 2 shows how the safety condition for P_0 is translated.

Iteration ...
0 states the safety formula for P_0;
1 replaces the 1st communication subformula by q_{01} and defines q_{01} for P_1;
2 replaces the 2nd communication subformula by q_{02} and defines q_{02} for P_2.

The translation can be iterated for P_1 and P_2 respectively.

iteration	object	local formulae	communication
0	P_0	$\mathsf{A}\,\mathsf{G}(crit_0 \Rightarrow P_1.\neg crit_1 \wedge P_2.\neg crit_2)$	
1	P_0	$\mathsf{A}\,\mathsf{G}(crit_0 \Rightarrow q_{01} \wedge P_2.\neg crit_2)$	$q_{01} \Rightarrow P_1.q_{01}$
	P_1	$q_{01} \Leftrightarrow @P_0 \wedge \neg crit_1$	$q_{01} \Rightarrow P_0.q_{01}$
2	P_0	$\mathsf{A}\,\mathsf{G}(crit_0 \Rightarrow q_{01} \wedge q_{02})$	$q_{02} \Rightarrow P_2.q_{02}$
	P_2	$q_{02} \Leftrightarrow @P_0 \wedge \neg crit_2$	$q_{02} \Rightarrow P_0.q_{02}$

Fig. 2. P_0 safety condition transformed to D_0

Note that new action symbols are introduced along with the translation. However, "new" in this context means that the symbols have not been in the formula before but may very well have been in the object signature. In practice where we have implemented the system and want to check assertions in retrospect, this means that we have to look for suitable communication symbols that are already implemented. In this sense, the translation provides the "communication requests" that must be matched by the communication as modelled in system design. We elaborate on this idea in the next section.

3 Verifying Checking Conditions

The method for translating D_1 to D_0 formulae sketched above, generates new communication symbols as shown in figure 2.

In general, we have to find a mapping of the generated communication formulae α, β to the modeled action calling terms of the model. This operation can be automatised when combined with model checking in the following way. Recall the above D_1 formula $\varphi \Leftrightarrow i.(\ldots j.\psi \ldots)$ and the local D_0 formula $\delta \Leftrightarrow j.(q \Leftrightarrow @i \wedge \psi)$ which was generated during the translation. Note that we use the transformed transition model, in which the labels of the states include action symbols as state predicates. Now we compute by model checking the set S' of states of object j in which ψ holds. Let $B_j^{S'}$ be the set of action symbols that are in the labels of each state in S'. In order to compute the possible set of actions that imply the

formula ψ in j, we have to remove from $B_j^{S'}$ the set of action symbols that occur in any label of states not in S'. We call this set B_j^{Ac}, the set of actions that are potentially able to communicate with i preserving the formula ψ.

The above formula δ is true for all action symbols of B_j^{Ac} establishing a communication between object i and object j. Let $B_j^{Ca} \subseteq B_j^{Ac}$ be the set of communication actions of object j that perform a communication with object i and satisfy ψ. Note that the set B_j^{Ca} is easily computed by intersecting the set of actions of the modeled action callings of object j with B_j^{Ac}: every action of B_j^{Ac} that occurs in the action calling terms of object j, is a communication action and thus belongs to the set B_j^{Ca}. The set B_j^{Ca} denotes the set of actions that perform a communication with i preserving ψ.

After checking this we have to check the generated D_0 formula $\overline{\varphi} \Leftrightarrow i.(\ldots q \ldots)$ for all $q \in B_i^{Ca}$. The set B_i^{Ca} includes all communication actions q of object i that call an action $\mathrm{b} \in B_j^{Ca}$ in j. Instead of checking $\overline{\varphi}$ for all actions q in B_i^{Ca} we replace q by the disjunction of all actions in B_i^{Ca}. By finding at least one communication action satisfying $\overline{\varphi}$, a ψ preserving communication between i and j can be established. Note that checking of $\overline{\varphi}$ and δ can be done on the local models of object i and j; no global state transition graph is needed to verify these formulae.

```
Check-D0(φ̄, ψ: D0 formula; i, j: I): bool ;
  begin
    S' := Compute_States (j, ψ);
    B_j^S' := ⋃ {a | a = L_|Ai(s), s ∈ S'};
    B_j^Ac := B_j^S' − {a | a = L_|Ai(s), s ∈ S̄'};
    B_j^Ca := B_j^Ac ⋂ {a | j.(a ⇒ i.b)};
    B_i^Ca := {b | i.(b ⇒ j.a), a ∈ B_j^Ca};
    replace (φ̄, q → ⋁ B_i^Ca);
    return Mod_Check (i, φ̄);
  end ;
```

Fig. 3. Algorithm for verifying D_0 formulae

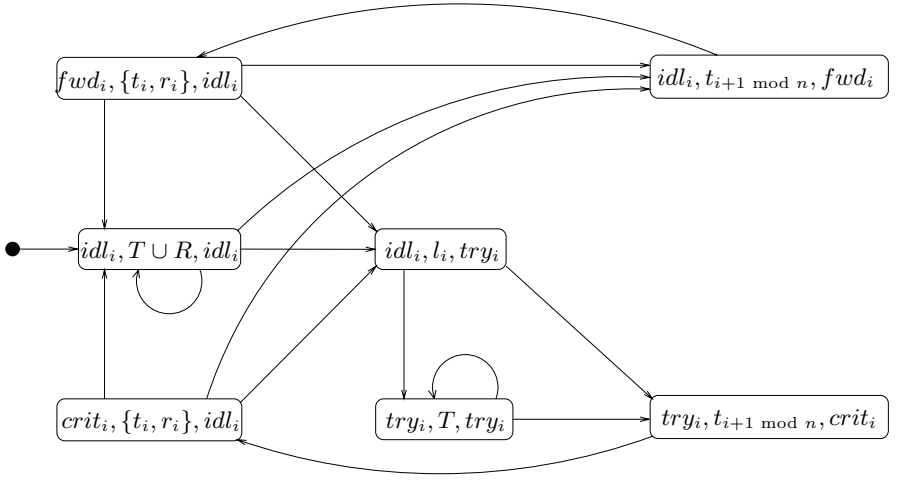
Figure 3 gives an algorithmic notation of the algorithm outlined above. The procedure Compute_States is a subroutine that is usually used in model checkers for computing the set of states satisfying a given formula. The replace procedure replaces the generated action symbol q with the disjunction of all action symbols of set B_i^{Ca}. If the set B_i^{Ca} is empty, then q would be false. The function $L_{|A_i}(s)$ is an abbreviation for $L(s) \cap A_i$. The procedure Mod_Check can be seen as a usual model checking procedure.

The overall truth value of the global D_1 formula depends on the truth values of the local D_0 formulae $\overline{\varphi}$ and δ and the correct mapping of the generated communication formulae α and β with the modeled communication requests, which is done by computing the sets B_i^{Ca} and B_j^{Ca}. It may happen that one of the above action sets is empty; that would be the case if there is no state in j satisfying ψ or if there is no suitable mapping from the computed actions

to modeled actions. In this case no communication between i and j can be established that satisfies ψ in j; the generated action formula q is replaced by false to indicate that there is no ψ preserving communication to j. If so, detailed warnings should be given to the user helping to find the error in the design.

By iterating the above algorithm for every D_1 communication subformula from the innermost communications to the outermost ones, the successful computation of S' and thus a valid procedure call for the model checker is guaranteed.

Example 3 (Mutual Exclusion cont.). To illustrate the above algorithm, we first have to translate the original model to the model with actions as state predicates. Figure 4 shows the model of process i after the translation. The value of the labeling function L_i is denoted by the second and the third item of each state label. The notation $\{t_i,\ r_i\}$ as a label in the states is an abbreviation for two states; usually every state has exactly one transition in the labelling. Correspondingly, the notation $T \cup R$ is also an abbreviation.



$$T = \{t_j \mid j \neq i + 1 \bmod n\}, R = \{r_k \mid k \neq i\}$$

Fig. 4. Translated model for the mutual exclusion example

Before giving an example for the algorithm, we refer to the D_0 specification generated in the previous section (figure 2). Iteration 1 produces the new action symbol q_{01}. The occurrence of symbol q_{01} should be equivalent to the formula $@P_0 \wedge \neg\, crit_1$ in process P_1. Following the algorithm we have to compute the set of states satisfying $\neg\, crit_1$ in P_1. This is true for all states except for $(try_1, t_2, crit_1)$. In the next step we compute the set of actions that are in the label of all states in S', obtaining $\{t_0, t_1, t_2, r_0, r_1, r_2, l_1\}$. By computing the set of potential communication actions, we have to reduce this set by all actions that are not in the labels of states of S'. In our example, t_2 is the only action that occurs outside of S'.

P_0	P_1	P_2
$P_0.(t_0 \rightarrow P_1.t_0)$	$P_1.(t_0 \rightarrow P_0.t_0)$	$P_2.(t_0 \rightarrow P_0.t_0)$
$P_0.(t_0 \rightarrow P_2.t_0)$	$P_1.(t_0 \rightarrow P_2.t_0)$	$P_2.(t_0 \rightarrow P_1.t_0)$
$P_0.(t_1 \rightarrow P_1.t_1)$	$P_1.(t_1 \rightarrow P_0.t_1)$	$P_2.(t_1 \rightarrow P_0.t_1)$
$P_0.(t_1 \rightarrow P_2.t_1)$	$P_1.(t_1 \rightarrow P_2.t_1)$	$P_2.(t_1 \rightarrow P_1.t_1)$
$P_0.(t_2 \rightarrow P_1.t_2)$	$P_1.(t_2 \rightarrow P_0.t_2)$	$P_2.(t_2 \rightarrow P_0.t_2)$
$P_0.(t_2 \rightarrow P_2.t_2)$	$P_1.(t_2 \rightarrow P_2.t_2)$	$P_2.(t_2 \rightarrow P_1.t_2)$

Fig. 5. Action calling terms of processes P_0, P_1, P_2

Now we continue with the modeled action callings to produce the set of communication formulae satisfying $\neg\, crit_1$ in j. The action callings of P_1 are given in figure 5. The only actions that perform communications with P_0 and match with the computed set of communication actions are $\{t_0, t_1\}$. Thus we have to find action callings in P_0 that communicate with P_1 via actions t_0 and t_1 as well. Due to the fact that P_0 has suitable communication requests that match with the computed actions, we have found a mapping between the generated communication requests of figure 2 and modeled action callings of figure 5. The overall truth value of $\overline{\varphi}$ can be computed by replacing q_{01} by the communication requests found by the algorithm. Repeating the above procedure for q_{02} we get the formula $\overline{\varphi} \;\Leftrightarrow\; \mathsf{A}\,\mathsf{G}\;\; (crit_0 \;\Rightarrow\; (t_0 \vee t_1) \wedge (t_1 \vee t_2))$. The truth of this formula is obvious in this case.

object	local formulae	communication
P_0	$\mathsf{A}\,\mathsf{G}(crit_0 \;\Rightarrow\; t_1)$	$t_1 \Rightarrow P_1.t_1$
		$t_1 \Rightarrow P_2.t_1$
	$t_2 \Leftrightarrow @P_1 \wedge \neg\, crit_0$	$t_2 \Rightarrow P_1.t_2$
	$t_0 \Leftrightarrow @P_2 \wedge \neg\, crit_0$	$t_0 \Rightarrow P_2.t_0$
P_1	$\mathsf{A}\,\mathsf{G}(crit_1 \;\Rightarrow\; t_2)$	$t_2 \Rightarrow P_2.t_2$
		$t_2 \Rightarrow P_0.t_2$
	$t_0 \Leftrightarrow @P_2 \wedge \neg\, crit_1$	$t_0 \Rightarrow P_2.t_0$
	$t_1 \Leftrightarrow @P_0 \wedge \neg\, crit_1$	$t_1 \Rightarrow P_0.t_1$
P_2	$\mathsf{A}\,\mathsf{G}(crit_2 \;\Rightarrow\; t_0)$	$t_0 \Rightarrow P_0.t_0$
		$t_0 \Rightarrow P_1.t_0$
	$t_1 \Leftrightarrow @P_0 \wedge \neg\, crit_2$	$t_1 \Rightarrow P_0.t_1$
	$t_2 \Leftrightarrow @P_1 \wedge \neg\, crit_2$	$t_2 \Rightarrow P_1.t_2$

Fig. 6. Local safety conditions and communication after matching

After iterating the above method for P_1 and P_2 we obtain a set of local safety conditions and action calling formulae as shown in figure 6. In the case of our example the algorithm computes a matching of the generated communication symbols $q_{i(i+1 \bmod 3)}$ and $q_{i(i-1 \bmod 3)}$ with the modeled actions $t_{i+1 \bmod 3}$ for $i \in \{0, 1, 2\}$ taken from the action calling terms.

With the matching of the generated communication symbols with the modeled action callings we are able to satisfy the local D_0 formulae (if possible). Due to the sound and complete translation of a given D_1 formula to D_0 formulae (cf. [EC00]), the truths of the global D_1 formula is implied by the truths of the local

D_0 formulae. Once we have found a matching satisfying the local D_0 formulae, the global D_1 formula is satisfied as well. Thus our reasoning from the local to the global level is sound due to the soundness and completeness of the D_1 to D_0 translation.

4 Concluding Remarks

The investigation of the idea presented in this paper is at its beginning, further study is necessary in order to explore how the idea can best be utilised in checking object system designs. For doing this, appropriate tool support in combination with existing model checkers is essential.

Although we introduce new communication conditions, the overall verification is promised to be very efficient. Due to the fact that the model checking complexity grows linearly with the length of a given CTL-formula (cf. [CGP00]), the local model checking amount is increased linearly by introducing new communication conditions. Compared to the verification of global models, the extra effort needed for checking newly introduced communication conditions on local models should be relatively small.

The technique described here has the limitation that we are able to deal with direct communication only. Note that there must be a direct communication between two components i and j, if e.g. i uses a communication subformula of observer j. Systems using indirect or asynchronous communication can not be verified directly: whereas indirect communication has to be split into its direct communication parts on the specification level, systems using asynchronous communication have to be transformed to synchronous ones. The transformation might be done by using special buffer objects.

The original motivation for studying D_1 and D_0 and their translation was to use them for modelling and specification [ECSD98]. D_0 was chosen to describe the semantics of the TROLL language, which was developed along with an application project to design an information system for a laboratory in the German National Institute of Weights and Measures (PTB) in Braunschweig, cf. [HDK+97,GKK+98].

References

[AL93] Martin Abadi and Leslie Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[AL95] Martin Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. *Lecture Notes in Computer Science*, 131:52–71, 1981.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doran A. Peled. *Model Checking*. MIT Press, 2000.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional Model Checking. In *Proceedings fo the 4th Annual Symposium on Principles of Programming Languages*, pages 343–362, 1989.

[DH97] G. Denker and P. Hartel. TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics. Informatik-Bericht 97–03, Technische Universität Braunschweig, 1997.

[dNV90] Rocco de Nicola and Frits Vaandrager. Action versus State based Logics for Transition Systems. In I. Guessarian, editor, *Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419, 1990.

[dR97] Willem-Paul de Roever. The Need for Compositional Proof Systems: A Survey. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 1–22, September 1997.

[EC00] H.-D. Ehrich and C. Caleiro. Specifying communication in distributed information systems. *Acta Informatica*, 36 (Fasc. 8):591–616, 2000.

[ECSD98] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for Specifying Concurrent Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer Academic Publishers, 1998.

[EP00] H.-D. Ehrich and R. Pinger. Checking object systems via multiple observers. In *International ICSC Congress on Intelligent Systems & Applications (ISA'2000)*, volume 1, pages 242–248. University of Wollongong, Australia, International Computer Science Convetions (ICSC), Canada, 2000.

[FM95] Jose Luiz Fiadeiro and Tom Maibaum. Verifying for Reuse: Foundations of Object-Oriented System Verification. In C. Hankin, I. Makie, and R. Nagarajan, editors, *Theory and Formal Methods*, pages 235–257. World Scientific Publishing Company, 1995.

[GKK+98] A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger, and H.-D. Ehrich. The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools. In T.W. Ling, S. Ram, and M.L. Lee, editors, *Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore*, pages 277–290. Springer, LNCS 1507, November 1998.

[GL94] Orna Grumberg and David E. Long. Model Checking an Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[HDK+97] P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. Information systems modelling with TROLL formal methods at work. In *Information Systems*, volume 22, pages 79–99, 1997.

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science - Modelling and reasoning about systems*. Cambridge University Press, 2000.

[Jos89] Bernhard Josko. Verifying the Correctness of AADL Modules using Model Checking. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 386–400, 1989.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, July 1981.

[Pnu85] Amir Pnueli. In Transition From Global to Modular Temporal Reasoning about Programs. In Krzysztof R. Apt, editor, *NATO ASI Series*, volume F13, 1985.