Journal of Universal Computer Science, vol. 9, no. 2 (2003), 106-119
submitted: 14/10/02, accepted: 14/2/03, appeared: 28/2/03 © J.UCS

Checking Object System Designs Incrementally

Hans-Dieter Ehrich
(Technical University Braunschweig
HD.Ehrich@tu-bs.de)

Maik Kollmann
(Technichal University Braunschweig
M.Kollmann@tu-bs.de)

Ralf Pinger
(SIEMENS AG Transportation Systems
Ralf.Pinger@siemens.com)

Abstract: We present a method for checking global conditions for object systems in a way that avoids state space explosion. The objects referred to in a global condition are checked step by step against local conditions and communication requirements derived from the global condition. The derivation is automatic, based on information about the system structure contained in the global condition. The approach is demonstrated using model checking, but the idea works for other approaches to verification or testing as well. In our current investigation, a multi-object variant of CTL is used for expressing global conditions. The local conditions and communication requirements can be verified independently using standard model checkers. The method is illustrated by a large example (about 10^{24} states) where our method shows a spectacular speedup over global model checking.

Key Words: multi-object logic, model checking, modelling and design, object system, temporal logic, verification.
Category: F.3.1, I.6.4

1 Introduction

By an object system, we mean a community of sequential objects operating concurrently and communicating synchronously in an RPC-like fashion. This is in accordance with current middleware technology.

When checking such systems using automatic verification or testing methods, the complexity tends to grow exponentially with the number of objects: the state space "explodes" in size. This puts very tight limits on the practicality of checking object systems.

We show that these limits may be overcome if information about the system structure is available at the time a global condition is written. The idea is to reflect the system structure in the global condition in a way that enables automatic translation to local conditions plus communication requirements. Then, instead of checking the global model which is the product of the local models, a network of communicating local models is checked incrementally. The checking involves only the models referred to in the condition. The savings in overall effort may be spectacular.

Our method looks set to enable verification of large object systems in an early phase of development, using affordable standard equipment and not requiring too much human expertise. This may bring checking and verification methods into broader practical use. This in turn may help to improve software quality and to avoid costs which arise if errors are detected late in the software development life cycle.

Moreover, the work can be split and parallelized easily, letting the local checks be performed on different machines. And it is possible to check incomplete designs as soon as models of those objects are available to which the global conditions in question refer.

The method put forward here does not directly compare with compositionality approaches [Pnu85, CLM89, GL94, dR97]: these work on the grounds that no knowledge about the system's structure is available when writing the global conditions. Consequently, there is the crucial step of recognizing internal interfaces and equipping these with appropriate assumptions and guarantees. This requires much human expertise and cannot be automatized except for very restricted cases. Similar arguments hold for abstraction techniques. [CGP00] gives an overview of these techniques and others. [Pin02] gives a detailed comparison with our approach.

There is another fully automatic state-reduction technique in use: partial-order reduction [KP88, CGP00]. It does not easily compare with our method: it is restricted to interleaving composition and linear-time temporal logic without the next operator. Our approach doesn't have these limitations, so we decided to demonstrate it in cases where partial-order reduction does not apply. A detailed comparison is under study. For the moment being, we use global model checking as a yardstick for comparison where the global state space is constructed and no reduction technique is applied.

RPC-like object interaction cannot be directly expressed in current model checkers, but it can be encoded in a rather straightforward way. We do not elaborate on this aspect here and refer the reader to [Pin02] for details.

Our approach is based on using a 'global' logic D_1 for expressing multi-object conditions that is automatically translatable to a 'local' logic D_0 for expressing single-object conditions and communication requirements [ECSD98, EC00]. While the original idea was to use this for object system specification, the approach was later elaborated for model cheking. [EP00] and [PE01] contain former versions of the basic idea illustrated with small examples, and [Pin02] reports on refining and implementing the method and running big experiments. The current paper communicates results of that work.

An explanation is in order what we mean by 'global' and 'local' in our logics. The problem is how to express properties of a set of sequential objects working concurrently. The temporal logics having well-developed verification tools can only express properties of sequential systems. Conventionally, a global property referring to several objects is interpreted in a global sequential system composed from the objects in question. This composite system may explode in size. In contrast, we leave the objects separate and associate the logics with them, avoiding to build the global state space. Consequently, all our conditions are local—strictly speaking. For expressing 'global' properties, we express statements about other objects in a local logic. Thus, by a 'global' statement we mean a statement which contains substatements referring to other objects. The validity of these substatements is evaluated on the basis of synchronous communication during which two objects are coupled so tightly that they act as one sequential system. This coupling, however, is volatile and only implicit.

For the work reported here, we used CTL, the computation tree logic proposed by E. Clarke and A.E. Emerson [CE81], and the SMV model checker [McM96]. Folklore says that, while CTL is more efficient in model checking, LTL is more intuitive for humans. We cannot confirm the latter, though, after working through a number of examples. We demonstrate the practicality of the method by model checking a large example (about 10^{24} states) which took a few minutes with our approach while global model checking took hours on the same equipment. In order to make it easier to compare with other approaches, we chose the steam boiler example which is well known from the verification literature (cf. [ABL96]).

Current model checkers like SMV offer module concepts where each module represents a process which may be abstractly characterized as a Kripke structure [Kri63a, Kri63b]. Two modes of cooperation are usually supported: step-by-step synchronization of all modules using a global clock, and interleaving by taking one step of one module at a time. In contrast, our approach supports what was called "perspective concurrency" in [Pin02], allowing objects to perform transitions independently of each other. It may then depend on the viewpoint of a sequential observer whether steps of different objects appear synchronous or in any sequence.

In the next section, we give a brief informal introduction to our approach; details can be found in [Pin02].

2 Multi-object logics

Temporal logics like CTL are designed for describing properties of sequential systems: a path formula refers to a given trace of states, and a state formula refers to a given state, possibly stating a property of the collection of paths emanating from that state. Using temporal logic for a communicating network of concurrent objects requires some adaptation: we use CTL locally for the sequential objects, and combine the local logics via communication constructs. The latter may be done in different ways, and here is where our logics D_0 and D_1 differ.

Let I be a finite set of identities representing sequential objects. Misusing terminology slightly, we speak of "object i" when we mean the object with identity i. Each object $i \in I$ has an individual set P_i of atomic state predicate symbols, its *signature*. In applications, predicate symbols may express which attributes have which values, which actions are enabled, which actions have occurred, etc. For the purpose of this paper, we need not go into detail here.

We use the multi-object logic D_1 as described in [ECSD98, EC00] but instantiated with CTL. This means that we have a local logic i.CTL over signature P_i for each object $i \in I$. D_1 allows to use formulae from j.CTL for any other object $j \in I$ as subformulae within i.CTL. These constituents are called *communication subformulae*.

In anticipation of the steam boiler example in section 5, let us look at simplified examples of conditions for a system consisting of three objects: control, boiler, and pump. We want to express the control rule that if the control is in normal mode, the water level is ok and the steam volume in the boiler gets high, then the pump is switched on; if the pump does not work, control leaves normal operation.
This condition is best attached to the control but refers to the boiler and the pump, therefore it is considered global.

$$\begin{aligned}&\text{control.}\mathsf{AG}(\ \text{mode} = \text{normal} \wedge \text{water-level} = \text{ok} \wedge \text{boiler.(steam-level} = \text{high)}\\ &\qquad\qquad \Rightarrow \mathsf{AX}\text{pump.(action} = \text{geton)} \vee \mathsf{AF}\ \text{mode} \neq \text{normal})\end{aligned}$$

The following condition is local because it does not refer to any other object. It says that the pump should start operation if the corresponding action is called, unless it is defective or in its stop state.

$$\text{pump.}\mathsf{AG}(\text{state} \neq \text{stop} \wedge \text{state} \neq \text{defective} \wedge \text{action} = \text{geton} \ \Rightarrow \text{state} = \text{on})$$

Please note that the propositions are in different local logics: 'mode = normal' is local to the control, 'steam-level = high' applies to the boiler, 'action = geton' makes only sense for the pump, etc.

Formally, the syntax of $\mathsf{D}_1 = \{D_1^i\}_{i \in I}$ is given by the following BNF grammar. We make use of well-known adequate sets of propositional connectives and of CTL operators (i.e., all others can be derived). In addition to P_i, we have propositional symbols $@j$ in i's local logic, for every $j \in I$; these are supposed to mean that i currently synchronizes with j. This is redundant because we will have $i.(@j \Leftrightarrow j.\ \mathit{true})$, but we introduce it for the sake of compatibility with D_0, see below.

$$D_1^i ::= i.\Psi_1^i$$
$$\Psi_1^i ::= P_i \mid @I \mid \neg\Psi_1^i \mid (\Psi_1^i \vee \Psi_1^i) \mid \mathsf{EX}\,\Psi_1^i \mid \mathsf{EG}\,\Psi_1^i \mid \mathsf{E}[\Psi_1^i\,\mathsf{U}\,\Psi_1^i] \mid C_1^i$$
$$C_1^i ::= \ldots \mid D_1^j \mid \ldots \qquad (j \in I, j \neq i)$$

The intended meaning of D_1 is that, in a formula $i.(\ldots j.\psi\ldots)$, the communication subformula $j.\psi$ holds in a given state of i iff i synchronizes with j in a state where ψ holds during this synchronization. Formal interpretation of D_1 is given in the next section.

D_1 has an interesting sublogic called D_0: it is like D_1 without communication subformulae but with separate communication formulae of the kind $i.(p \Rightarrow j.q)$ where $p \in P_i$ and $q \in P_j$ are propositional symbols. These communication formulae mimic RPC as supported by current middleware: in i, p *(a call)* implies synchronization with j assuring q *(execution of the called procedure)* there (possibly moving data back and forth during synchronization—but this is not reflected in the abstraction).

As an example, the first of the above conditions can be stated in D_0 as follows.

control.AG(mode=normal $\wedge$ water-level=ok $\wedge$ q_1 $\Rightarrow$ $\mathsf{AX}\,r_1$ $\vee$ AF mode$\neq$normal)
boiler.(q_2 $\Rightarrow$ @control $\wedge$ steam-level = high)
pump.(r_2 $\Rightarrow$ @control $\wedge$ action = geton)
control.(q_1 $\Rightarrow$ boiler.q_2)
boiler.(q_2 $\Rightarrow$ control.q_1)
control.(r_1 $\Rightarrow$ pump.r_2)
pump.(r_2 $\Rightarrow$ control.r_1)

The q's and r's are communication predicates establishing the obvious synchronization.

Formally, the syntax of D_0 is given by the following BNF grammar.

$$D_0^i ::= i.\Psi_0^i \mid i.C_0^i$$
$$\Psi_0^i ::= P_i \mid @I \mid \neg\Psi_0^i \mid (\Psi_0^i \vee \Psi_0^i) \mid \mathsf{EX}\,\Psi_0^i \mid \mathsf{EG}\,\Psi_0^i \mid \mathsf{E}[\Psi_0^i\,\mathsf{U}\,\Psi_0^i]$$
$$C_0^i ::= \ldots \mid (P_i \Rightarrow j.P_j) \mid \ldots \qquad (j \in I, j \neq i)$$

In applications, not all propositional symbols in P_i would probably be acceptable as communication symbols, a subset $\mathcal{A}_i \subseteq P_i$, e.g., of action occurrence symbols, would be used instead. We refrain from elaborating on this ramification.

3 Interpretation

We give a brief account of D_1 interpretation (cf. [EC00]). This includes an interpretation of D_0 since it is a sublogic of D_1. We assume that the reader is familiar with CTL interpretation over Kripke models $\mathcal{M} = (S, S_0, \rightarrow, L)$ as described, e.g., in [HR00]. S is a set of states with subset S_0 of initial states, $\rightarrow \subseteq S \times S$ is the transition relation, and $L : S \to 2^P$ is the state labelling function associating a set of propositional symbols with each state. For simplifying interpretation definition, we assume $\to$ to be total, i.e., every state has at least one next state. Thus, there are no finite life cycles (maximal traces).

In our setting, we assume a family $\{\mathcal{M}_i\}_{i \in I}$ of models, one for each object. Conventional interpretation of CTL formulae defines the meaning of $\mathcal{M}_i, s_i \models_i \varphi$ for every object $i \in I$ where $s_i \in S_i$ and $\varphi \in \mathsf{D}_1^i$, as long as φ does not contain a communication subformula.

For capturing communication, we note that we have to synchronize at states across models. However, a state of an object may be reentered several times during a trace, and it need not synchronize each time with the same state of another object. For instance, in a mutual exclusion example, if a process leaves the critical region, it may synchronize with another process to enter if it wants to. If the latter process is in its ready state, it will enter; if it is in its idle state, it will not. If the first process leaves the critical region several times during its trace, the second process will enter in some cases and not in others. The following definition captures this intuition.

$$\begin{array}{lll} \mathcal{M}_i, s_i \models_i j.\psi & \text{iff} & \mathcal{M}_i \text{ in } s_i \text{ synchronizes with some state } s_j \text{ in } \mathcal{M}_j, \\ & & \text{and for every state } s_j \text{ in } \mathcal{M}_j \text{ with which } \mathcal{M}_i \text{ in} \\ & & s_i \text{ may synchronize, we have } \mathcal{M}_j, s_j \models_j \psi \end{array}$$

This means that i synchronizes with j whenever i enters state s_i, and all states in j which may possibly synchronize with s_i must satisfy ψ.

More details can be found in [EC00] where interpretation of D_1 is given more comprehensively in terms of event structures (albeit for a different local logic).

4 Localizing global conditions

In [EC00], a sound and complete translation $\mathsf{D}_1 \to 2^{\mathsf{D}_0}$ is presented: working inside-out, every formula $\varphi \Leftrightarrow i.(\ldots j.\psi \ldots)$ with an innermost communication subformula $j.\psi$ is replaced by the D_1 formula $\varphi' \Leftrightarrow i.(\ldots\ q_i\ \ldots)$ and the D_0 formulae $\delta \Leftrightarrow j.(q_j \Leftrightarrow @i \wedge \psi)$, $\alpha \Leftrightarrow i.(q_i \Rightarrow j.q_j)$, and $\beta \Leftrightarrow j.(q_j \Rightarrow i.q_i)$, where q_i and q_j are propositional symbols to be matched with existing ones in the signatures P_i and P_j, respectively (see below). The formulae α and β are called *communication requirements.*

The D_1 and D_0 examples in section 2 give an example of such a translation. Note that the number of communication subformulae in φ' is one less than that in φ. If φ' is not yet in D_0, the transformation step is iterated for another innermost communication subformula. Since the number of communication subformulae is finite and strictly decreases in each step, the transformation terminates after a finite number of iterations.

This transformation can be used to break global D_1 checking conditions down into sets of D_0 conditions which can be checked locally, and communication symbols which have to be matched with existing ones according to the communication requirements. This is elaborated in [EP00, PE01, Pin02].

The matching algorithm follows the translation steps $\varphi \Leftrightarrow i.(\ldots j.\psi \ldots) \longrightarrow \{\varphi', \delta, \alpha, \beta\}$ as given above. Here is a rough sketch. For each translation step, the following actions are performed.

1. Compute the set S_ψ of states of object j in which ψ holds. This is done by model checking.

2. Retrieve the set R^i_ψ of all communication symbols $r_j \in P_j$ that occur in a subset of S_ψ and establish communication with i (i.e., the D_0 communication formulae $j.(r_j \Rightarrow i.r_i)$ and $i.(r_i \Rightarrow j.r_j)$ hold for some $r_i \in P_i$).

   In other words, R^i_ψ is the set of all communication symbols in j having corresponding 'partner' symbols in i such that α, β, and δ as given above hold true.

3. Let Q^i_ψ be the set of communication symbols $q_i \in P_i$ that are in α-β correspondence with symbols $q_j \in P_j$; let

$$\varphi' \;\equiv\; i.(\ldots \textstyle\bigvee_{q \in Q^i_\psi} q \ldots).$$

φ holds iff φ' does. If φ' has no further communication subformulae, it can be locally model checked. Otherwise, the above step is repeated.

It may happen that one or the other of the above sets of communication symbols is empty; that would be the case if there is no state in j satisfying ψ, or if there is no suitable match of communication symbols. In this case, the disjunction in φ' above is empty and evaluates to *false*, indicating that there is no ψ preserving communication to j. If so, detailed warnings may be given to the user helping to find the error in the design.

More details can be found in [Pin02] where also the correctness of the algorithm is proved. Most interestingly, all checkings necessary for establishing global validity of the checking condition can be done on the local models of the objects. Standard model checkers are used not only for local model checking but also for essential parts of matching the communication symbols and checking the communication requirements.

5 The steam boiler example

We illustrate the practicality of the method with the steam boiler example as treated in [ABL96] (cf. figure 1). Duval and Cattel [DC96] have a model checking approach there using SPIN 2.8.5 with partial-order reduction enabled. This version is no longer available. We tried SPIN 3.2.0 but had to abandon the experiment after several days computing time because of lack of memory[1]. The same happened when we tried the global model with SMV. So we used a reduced example omitting, e.g., some of the technical error handling for water level and steam outlet control.

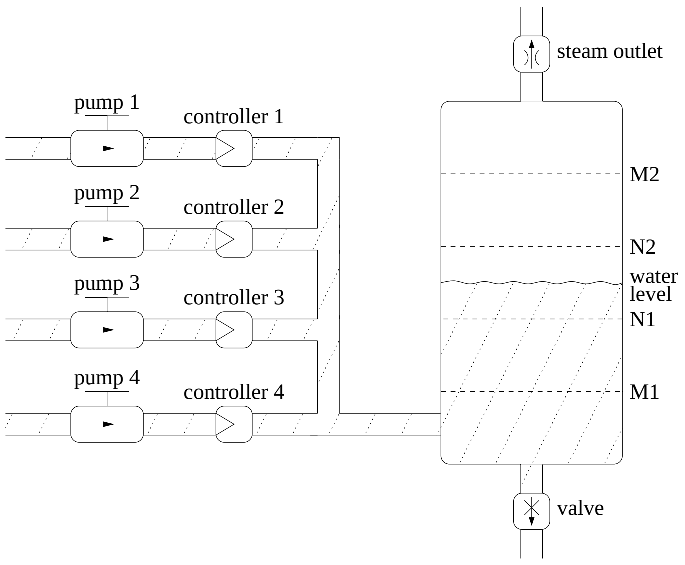

Figure 1: Steam boiler

First we recall structure and function of a steam boiler as presented in [ABL96]. Figure 1 shows the essential parts—with one exception: control. Otherwise, the boiler consists of a tank with four markings representing water levels. Between levels N1 and N2, the boiler works under normal condition. Outside this area but between levels M1 and M2, the boiler works in restricted mode.

In normal mode, the steam leaves the tank through the steam outlet at the top. The water level is adjusted via four pumps. Each of these is turned on or off by a controller.

If the water level is below M1, the boiler is about to overheat and switches off into stop mode. Likewise, the boiler stops if the level exceeds M2, indicating overpressure. Then the relief valve at the bottom opens in order to release water and let the level drop to normal.

The main task of the control is to keep the water level in normal mode,

[1] The experiment was run on a Sun UltraSPARC with 2 processors, 360 MHz, 2 GB memory each.

providing a safe operational system.

The steam boiler specification has been addressed and interpreted in various ways in the literature. There are treatments which concentrate on specification issues (e.g., [BHW96, BBD+96]), and others focussing on verification issues (e.g. [CW96, LL96]). Our approach follows the latter line of work. We checked the following set of conditions. It is the subset of conditions as laid down in [ABL96] that applies to our reduced model. We give the conditions in natural language and in multi-object logic D_1.

1. The valve is never open if control is in normal or restricted mode.

   $\text{control}.\mathsf{AG}(((\text{mode} = \text{normal} \vee \text{mode} = \text{restr}) \wedge @\text{boiler} \Rightarrow \text{boiler}.(\text{valve} \neq \text{open}))$

2. After the initialization phase, the water level is ok or control stops operation.

   $\text{control}.\mathsf{AG}(\text{mode} = \text{ini} \Rightarrow \mathsf{AF}\ (\text{boiler}.(\text{waterlevel} \geq \text{N1} \wedge \text{waterlevel} \leq \text{N2}) \vee \text{mode} = \text{stop}))$

3. In normal or restricted mode, the water level is never below M1 or above M2.

   $\text{control}.\mathsf{AG}((\text{mode} = \text{normal} \vee \text{mode} = \text{restr}) \wedge \text{clock} = \text{get-water-level} \Rightarrow \text{boiler}.\neg(\text{water-level} < \text{M1} \vee \text{water-level} > \text{M2}))$

4. If a pump is defective, it will resume normal operation when receiving a repair message.

   $\text{pump}_i.\mathsf{AG}(\text{state} = \text{defective} \Rightarrow \mathsf{AF}(\text{action} = \text{getrepaired} \Rightarrow \text{state} \neq \text{defective}))$

5. In normal mode of control, no pump is defective.

   $\text{control}.\mathsf{AG}(\bigvee_{i=1}^{4} \text{pump}_i.\text{state} = \text{defective} \Rightarrow \text{mode} \neq \text{normal})$

6. If, in normal mode of control and with water between normal levels, the steam volume goes up to 20, then all pumps are successively switched on or the control leaves normal operation because of pump defects.

   $\text{control}.\mathsf{AG}(\ \text{mode} = \text{normal} \wedge \text{water-level} = \text{N1N2} \wedge \text{clock} = \text{send-pump}_1 \wedge \text{boiler}.(\text{steam-level} = 20) \Rightarrow (\mathsf{AX}(\text{pump}_1.\text{action} = \text{geton} \wedge \mathsf{AX}(\text{pump}_2.\text{action} = \text{geton} \wedge \mathsf{AX}(\text{pump}_3.\text{action} = \text{geton} \wedge \mathsf{AX}\ \text{pump}_4.\text{action} = \text{geton}))))\vee \mathsf{AF}\ \text{mode} \neq \text{normal})$

   $\text{pump}_i.\mathsf{AG}(\text{state} \neq \text{stop} \wedge \text{state} \neq \text{defective} \wedge \text{action} = \text{geton} \Rightarrow \text{state} = \text{on})$

Table 1 shows the results, comparing the multi-object approach with the global approach (last column).

The speedup is spectacular. Condition 4 is especially interesting: the condition is local for pumps and requires only local checking of the pump state spaces in our approach. In contrast, the global approach does not exploit this knowledge and searches the global state space. Here we see that our approach automatically concentrates on the objects referred to in the conditions.

	multi-object				global
	control	boiler	pumps	total	total
#states	1.2×10^8	3.5×10^6	198		1.3×10^{24}
cond 1	$\approx$ 6 min 15 s	$\approx$ 42 s		$\approx$ 7 min	$\approx$ 2 h 57 min
cond 2	$\approx$ 4 min 38 s	$\approx$ 46 s		$\approx$ 5 min 20 s	$\approx$ 2 h 36 min
cond 3	$\approx$ 5 min 30 s	$\approx$ 46 s		$\approx$ 6 min 15 s	$\approx$ 2 h 52 min
cond 4			$<$ 1 s	$<$ 1s	$\approx$ 2 h 51 min
cond 5	$\approx$ 5 min 17 s		$<$ 1 s	$\approx$ 5 min 20 s	$\approx$ 2 h 50 min
cond 6	$\approx$ 4 min 27 s	$\approx$ 46 s	$<$ 1 s	$\approx$ 5 min 10 s	$\approx$ 2 h 43 min

Table 1: Results

6 Implementation details

The implementation of our method consists of three parts: a parser which transforms a D_1 formula into a syntax tree, a decomposer which translates this D_1 formula syntax tree into a set of D_0 formula syntax trees, and an interface which connects to the model checker.

The parser transforms a given D_1 formula into a syntax tree or stops if the formula is not in D_1. In our prototype, a module for CTL as the D_1 local logic is implemented. This module may be replaced by a module implementing LTL or ACTL or some other local logic.

The decomposer translates a D_1 formula into the set of D_0 formulae as explained above. Figure 2 illustrates the decomposition of checking condition 2 from the steam boiler example. The D_1 syntax tree is cut into as many pieces as the D_1 formula contains occurrences of object identifiers, starting at the root of the tree. Each time the decomposer reaches an innermost identifier, it introduces a new communication symbol which substitutes the rest of the tree. Additionally, a new tree for the inner identifier and its subtree is created. Along with this step, the corresponding communication requirements are generated (cf. figure 2).

The next step is to integrate the model checker. Our prototype implements an interface to the SMV model checker. So far, we do not make direct use of internal model checker interfaces which generate the set of states in which a checking condition φ holds. Instead, we call the model checker for every communication predicate, generating the set of matching predicates iteratively. Using internal interfaces which do this in one step would be an obvious optimization. The obvious drawback is that the model checker could not so easily be replaced by another one.

The prototype consisting of parser, decomposer and model checker interface

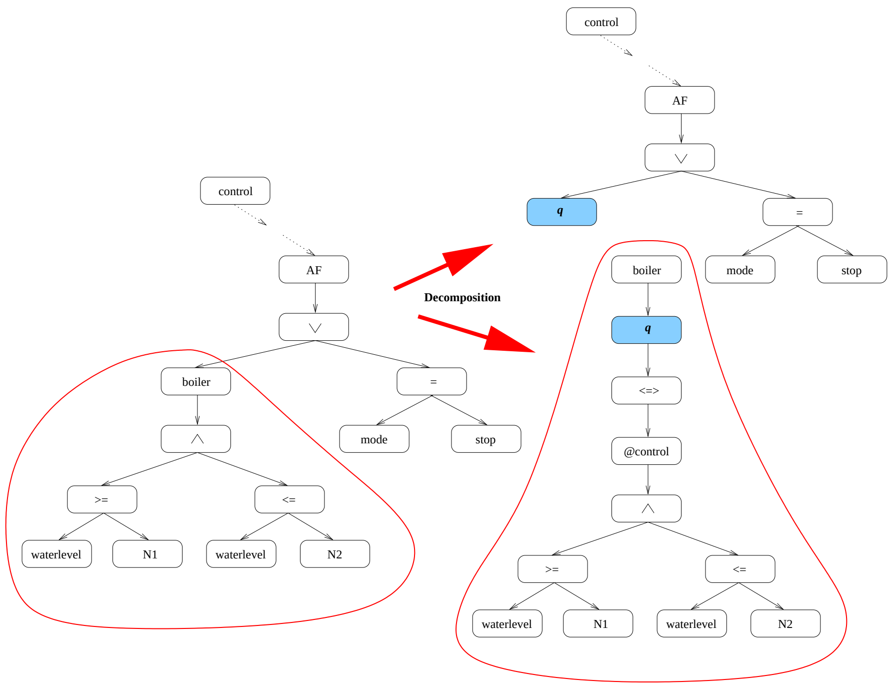

Figure 2: Decomposition of condition no. 2. The corresponding communication requirements: $\alpha \;\Leftrightarrow$ control.$(q \;\Rightarrow\; \mathit{boiler}.q)$, $\beta \;\Leftrightarrow$ boiler.$(q \;\Rightarrow\; \mathit{control}.q)$.

was developed in only one diploma thesis [Hör01]. This shows that it is not really much work to implement our method.

7 Concluding remarks

Further study will explore how the approach can best be utilized for checking object system designs and how it compares with other techniques like partial-order reduction. Appropriate tool support in combination with existing model checkers is essential. Tools for generating the SMV inputs from D_1 descriptions have been implemented [Pin02]. What would be welcome is better tool support for moving from statechart models of objects to the inputs of model checkers.

Because we incrementally check only the objects involved in the checking condition, the size of the entire system is not relevant for the method: it focusses automatically on the part of interest. Very big models can be checked as long as not too many objects are involved, those involved are small enough, and communication traffic between these is not too dense. In more precise complexity-theoretic terms, what we mean is that if the effort for checking an object and its communication requirements can be bounded by a constant, then the method is linear in the number of objects involved. It is good practice anyway to design systems in a way that constrains the size of objects and the amount of their interaction with other objects. And a single checking condition will in practice most often refer to a small part of the system only.

As mentioned above, we had to reduce the steam boiler example in order to get the global model through. This was necessary for comparing our technique with global model checking. However, we will implement the full model in order to demonstrate the practicality of our method also in cases where the model is too big for the conventional technique.

The fact that a 'global' D_1 checking condition in our approach is bound to an object and is expressed from a local viewpoint also brings pragmatic limitations: the condition can only express what can be 'observed' from this object in the sense that there is direct and synchronous communication. However, by nesting communication subformulae, complicated communication patterns may be expressed. There is not much experience yet, though, how useful and natural the logic is to express and check properties of objects not directly related by communication links. An example is the condition that a message sent by a sender will eventually be received by a receiver, but communication takes place via a transmission channel. In our method, there is no way to express the condition as a sender condition like "sender.(messages will eventually be received by the receiver)", mentioning the receiver but not mentioning the channel. On the other hand, it actually is a condition of the channel, and binding it to the channel enables to talk about sender and receiver in a natural way.

If there is no suitable object to bind the condition to, it is possible to define a new 'observer object' establishing just the communications that enable natural expression of the checking condition. This observer object, however, must faithfully mimic the behaviours in the objects involved, so we have to prove some form of equivalence. This may be too high a price to pay.

So far, we concentrated on systems with synchronous communication. Asynchronous communication may in principle be reduced to synchronous communication by introducing message buffers between any two communicating objects. However, this is not very elegant and hardly practical: if we treat the buffers as objects, the nesting depth of communication subformulae doubles, and the formulae get clumsy because the buffers have to be mentioned explicitly. We believe that there is a better solution, and finding it would be important for applying our method to checking widely distributed systems.

Acknowledgements

This work was partially supported by DFG under contract Eh 75/12 in the priority project SPP 1064.

References

[ABL96] Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[BBD$^+$96] Christoph Beierle, Egon Börger, Igor Dudanovic, Uwe Glässer, and Elvinia Roccobene. An evolving algebra solution to the steam boiler control specification problem. In Abrial et al. [ABL96].

[BHW96] Robert Büssow, Maritta Heisel, and Matthias Weber. A steam boiler control specification with statecharts and Z. In Abrial et al. [ABL96].

[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.

[CW96] Jorge Cuellar and Isolde Wildgruber. The Dagstuhl steam boiler controller problem: The TLT solution. In Abrial et al. [ABL96].

[DC96] Gregory Duval and Thierry Cattel. Specifying and Verifying the Steam Boiler problem with SPIN. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 203–217, 1996.

[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.

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

[Hör01] Daniel Hörnig. Ein Werkzeug zur Dekomposition von D_1-Spezifikationen. Master's thesis, Technische Universität Braunschweig, 2001.

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

[KP88] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Proc. Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency.*, volume 354 of *Lecture Notes in Computer Science*, pages 489–507. Springer-Verlag, 1988.

[Kri63a] S. A. Kripke. Semantical Analysis of Modal Logic I — Normal Modal Propositional Calculi. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Kri63b] S. A. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica — Modal and Many-valued Logics*, pages 83–94, 1963.

[LL96] Gunther Leeb and Nancy Lynch. Using timed automata for the steam boiler controller. In Abrial et al. [ABL96].

[McM96] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, second edition, 1996.

[PE01] R. Pinger and H.-D. Ehrich. Compositional Checking of Communication among Observers. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE), Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2001), Genova*, volume 2029 of *Lecture Notes in Computer Science*, pages 32–44, 2001.

[Pin02] R. Pinger. *Kompositionale Verifikation nebenläufiger Softwaremodelle durch Model Checking*. PhD thesis, Institut für Software – Abteilung Datenbanken, TU Braunschweig, 2002.

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