Assisting Groupware Development with Model Checking – Case Studies from Cooperative Work in the WWW

Constantinos Papadopoulos*

General Secretariat for Information Systems, Athens, Greece

Abstract: Efficient collaboration entails significant benefits for modern enterprises. Recent advances in Internet technology allow physically dispersed groups to bypass the obstacles raised by geographical distances, so the development of Internet based groupware may extend collaboration to a global scale. As groupware applications grow larger and more diverse, however, it becomes difficult to anticipate their correctness. In this paper, we address this difficulty within the context of group awareness, which we regard both as a communication issue and a user-interface one. The main contribution of our research is the use of symbolic model checking for verifying group awareness in collaborative work across the WWW. This involves the specification of two related protocols with temporal logic and the development of a methodology for encoding temporal formulae into the language of a symbolic model checker, which eliminates the need to draw state-transition diagrams. Taking then advantage of the model checker's ability to produce counterexamples, we discover drawbacks in these protocols and propose thereon a number of improvements, which aim to transform the WWW into a reliable collaborative environment.

Keywords: Computer-Supported Cooperative Work, World Wide Web, Group awareness, Requirements specification, Groupware verification, Temporal logic, Model checking.

1. INTRODUCTION

The World Wide Web (WWW) has become the common denominator for information exchange across the Internet and within enterprises. As such, it has drawn the attention of several research communities, including that of Computer Supported Cooperative Work (CSCW). CSCW has been defined as "*the set of computing applications that allow physically dispersed groups to engage in a common task by providing an interface to a shared workspace*" [23]. A primary goal in the design of such applications is to enable group members to maintain information on each other's presence and activities. This information is commonly referred to as *group awareness* [21] and is intended to facilitate the collaboration of spatially dispersed users. This is particularly important for the WWW, which was developed to support distributed workgroups in the first place [6].

Yet group awareness faces two challenges within that context. The first is related to the inheritance of a key Web property, which is the stateless nature of the HTTP protocol. In fact, since no information is maintained between successive HTTP requests, cooperative applications are often unaware of what page a client is currently browsing (lack of *workspace awareness*). The other challenge is that when an application changes, its users remain unaware of that change until they reload the application *via* another HTTP request (poor *activity awareness*). Such requests contribute however delays to the network and, as a result, applications involving frequent interactions cannot support group awareness adequately. Both these challenges indicate a need for better synchronization, which is inherently time dependent.

*Address correspondence to this author at the General Secretariat for Information Systems, Athens, Greece; E-mail: k.papadopoulos@gsis.gr

1.1. Web-Based CSCW

Various research efforts have addressed the above challenges through a number of prototypes, which incorporate collaborative features into the current infrastructure of the WWW. Exemplary among those prototypes are *Internet Foyer* [4], the protocol of Palfreyman and Rodden [44], *GroupScape* [27], *MetaWeb* [60], *Sideshow* [12], *MAUI* [31] and *F@*[59]. All these prototypes, however, provide solutions without guaranteed correctness. In fact, they are supposed to support distributed users who interact in different ways (i.e., synchronous/asynchronous), so the possibility of delays and interface inconsistencies is high. No one of these prototypes has been tested though against this possibility. Besides, errors in distributed and interactive computing are difficult to detect with traditional testing methods, due to the large number of event inter-leavings. Formal methods, instead, offer opportunities for automatic verification and, in addition to detecting errors they can also prove their absence in certain cases. Although CSCW is not yet a popular domain for formal methods, the few existing studies have yielded promising results.

1.2. Groupware Verification with Formal Methods

CSCW is an area that combines methods of software engineering, distributed computing, organization management and humancomputer interaction, and it involves thereof concepts which are defined vaguely. This entails that groupware developers provide often solutions which are difficult to understand and reason about. To counter this situation, some researchers have attempted to formalize CSCW and, based on this, to verify subsequently a number of systems [2, 20, 24, 26, 34, 35, 36, 37, 39, 43, 45, 46, 50, 54, 55, 56, 57, 58]. Nevertheless the support of awareness across the WWW has raised a number of issues, since network delays may cause inconsistencies on the interface and affect the *usability* of groupware applications [30].

1874-1339/07

Although some of the above works have addressed awareness from a user's perspective, the adequate support of awareness in Webbased collaboration is not ensured by any method (either formal or empirical).

1.3. Contributions

The research presented in this paper provides a framework for verifying properties of group awareness and improving certain aspects of groupware usability within the context of the WWW. Specifically, after expressing several requirements for awareness support that stem from actual collaboration practice, we specify then the behavior of two related protocols with temporal formulae and finitestate models. Based then on a methodology that we invented specifically for this purpose, we encode the temporal formulae into the *SMV model checker* [42] in order to simplify the encoding task and increase the efficiency of verification. By checking afterwards the encodings against the above requirements, we expose limitations in the two protocols and propose a number of improvements. The systematic refinement of these improvements later allows us to enhance the usability of the protocols, manifesting thus the soundness of our methodology. Another fact that manifests this soundness is that the encodings produced with our methodology are semantically equivalent with the encodings of the finitestate models, since the latter do not provide any additional information on the behavior of the protocols.

Other researchers contend that software systems can be better described with infinitestate models and verified then with *abstraction* techniques [7]. In this paper, instead, we provide evidence that certain properties of CSCW systems can be adequately described with temporal formulae without using statetransition models, and verified then with traditional methods of model checking. We must also point out that, although SMV was developed for verifying synchronous systems primarily, its use in the verification of asynchronous systems like the above two protocols is also quite feasible (in fact, SMV has been used previously in the verification of other asynchronous systems [48, 50].

Our approach is inherently iterative, as illustrated in Fig. (**1**) below:

The results of our research are significant for two reasons. First, group awareness depends on timing conditions that have profound implications for the usability of relevant protocols. To meet these conditions we suggest the infusion of correctness properties into the protocol semantics, which ensure the provision of awareness information in a consistent and timely fashion. Earlier work by other researchers has also employed formal methods for improving usability (e.g., the *IVY* project [36]), yet our own research focuses specifically on groupware protocols and demonstrates the efficiency of model checking for improving their usability. As we show later, usability can be expressed in some cases as a *fairness* requirement [25], indicating that if an event is possible it will happen eventually. Most important though, our encoding methodology makes model checking more attractive to groupware developers, since it saves them from the task of drawing statetransition models that may be painstaking for large systems.

2. BACKGROUND IN MODEL CHECKING

Model checking is an automatic verification technique, which compares formal specifications of desired properties against an abstract model of a system [18]. As concerns groupware systems, their correctness depends heavily on time, as the latter plays a critical role in cooperative work [3, 41]. For this reason, we have selected in this paper the formalism of *temporal logic* [17] to express the properties of the protocols that we wish to verify. This formalism is an extension of predicate calculus that can support reasoning on how the truth values of specifications change over time. Its rules are defined in terms of *states*, which assign values to system variables. Distinct states correspond to different valuations of those variables. A *transition* from one state to another occurs when the values(s) of one or more variable(s) in the first state change(s). An infinite sequence of states characterizing the execution of a process is called a *path*.

2.1. Temporal Logic as a Specification Language

Over the last three decades, temporal logic has been used to specify several kinds of *reactive systems*, whose role is to maintain an ongoing interaction with their environment [40]. Hence the correctness of these systems depends on synchronization properties and timing constraints. Groupware systems are multiuser reactive systems, so their specification must indicate the existence of alternative paths of synchronization. Consequently a *branchingtime logic* [16] is needed, which should comprise operators for the expression of relative event ordering and the quantification of events over paths. For the purpose of this paper we have

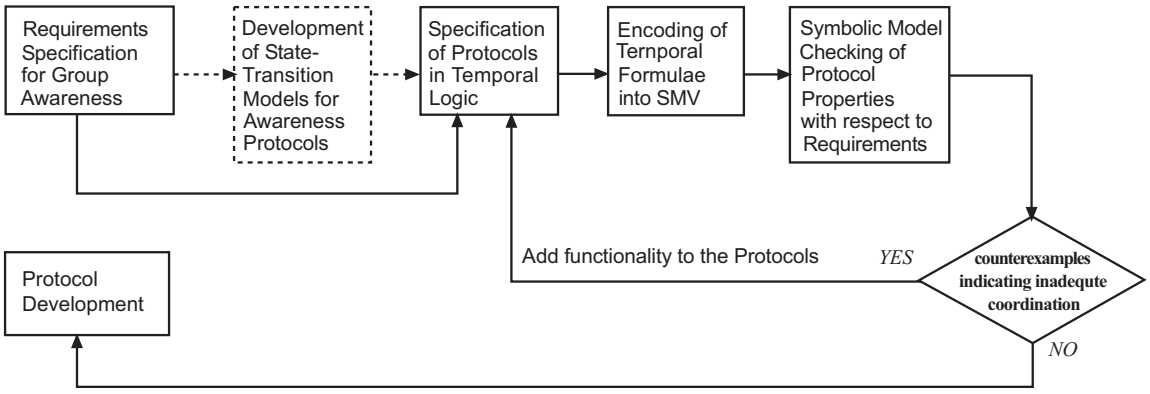

Fig. (1). Verification of groupware protocols with model checking.

chosen *computation tree logic* (CTL) [17], which is built of atomic propositions, logical connectives, and temporal operators. A formal definition of CTL is given in Section A.1 of the Appendix.

2.2. Model Checking Temporal Formulae

Within only a few years from its establishment, model checking turned out to be the prevailing method for verifying temporal formulae. When using this method, the system under consideration is modeled as a graph comprising a finite number of states, which are labeled by atomic propositions and connected by transitions. Graphs like this are known as *Kripke structures* and they are defined in this paper in Section A.2 of the Appendix. Model checking algorithms systematically verify whether a system satisfies a formula by searching every combination of paths in a Kripke structure and checking the formula's truth along the way. The total number of paths to be searched (and hence the complexity of these algorithms) depends on the number of free variables in the system's model. A particular advantage of model checking is that, when a system does not satisfy a given formula, it accompanies the negative answer with a counterexample falsifying that formula. Model checking algorithms are broadly classified into *explicit* and *symbolic* ones. The former operate on states and represent transition relations as adjacency lists. Because the number of states often becomes exponential in the number of parameters in the model, explicit algorithms suffer the *state explosion problem*. Symbolic algorithms, instead, operate directly on *symbolic* representations of state sets (i.e. representations based on Boolean formulae) and avoid thus the explicit enumeration of each state. A more detailed description of these algorithms is given in Section A.3 of the Appendix (as well as in [10] and [42]).

2.3. Managing State Explosion with Symbolic Algorithms

To make symbolic model checking practical, a method was needed to efficiently manipulate Boolean formulae like the above. A suitable method for this purpose is *binary decision diagrams* (BDDs) [9], which are like binary decision trees except that identical subtrees are merged and form directed acyclic graphs. BDDs can be further reduced to include each state variable at most once, by eliminating all redundant vertices whose edges point to the same vertex. Such diagrams hold essentially compressed forms of the truth tables of Boolean formulae, and thus symbolic algorithms explore several states at once instead of visiting one state at a time. Hence they can handle many orders of magnitude larger state spaces than explicit algorithms [10].

The first model checking tool that utilized symbolic algorithms was SMV, whose basic features are overviewed in Section A.4 of the Appendix. This tool extracts a BDD-based model from a system's encoding, which is given in a concurrent language. Programs in this language are annotated with CTL specifications, whose validity in the BDD model is checked by the symbolic algorithms of the tool. Whenever a model does not satisfy a specification, SMV produces a counterexample. Although BDDs incur theoretically a space complexity exponential in the length of Boolean formulae, their efficiency in capturing the execution patterns of reactive systems enables symbolic algorithms to exhibit time complexity that is linear in the number of states plus the length of the formulae being checked [9]. To verify large systems, however, symbolic algorithms require careful ordering of the input variables so as to avoid state explosion.

A complementary technique to BDD-based model checking is based on propositional *satisfiability* (SAT) and searches for counterexamples of an upper-bounded length [8]. More specifically, the transition relation of a system is unfolded k times, allowing thus any counterexample of up to k states to be found with a SAT solver. In the case that a bound on the length of counter-examples is not known, this method cannot verify a property but it can only produce counterexamples. Hence SAT-based model checking is referred to in the literature as an "*incomplete verification technique*" [47]. Whenever such a bound is known, however, model checking can be reduced to a satisfiability problem and bypass BDDs. In general, this method scales better with large systems than traditional symbolic model checking in terms of efficiency (i.e., execution time). More recently, Chechik *et al*. [15] introduced *multi-valued* symbolic model checking for formulae denoting uncertainty and inconsistency. This technique provides additional truth values to 0 and 1 but has the same complexity as CTL model checking.

Contrary to these works, in this paper we rely on the traditional algorithms of SMV to verify awareness protocols for the WWW. Like several other model checkers, SMV evaluates system variables in a next state relative to their values in the current state. Hence to verify a system with SMV, it is necessary to model it first with a state-transition diagram or an equivalent language. Regarding groupware systems though, this kind of modeling may be awkward and inefficient due to the large number of possible states. So in this paper we propose a methodology that encodes CTL operators directly into SMV, thus eliminating the need to draw state-transition diagrams and increasing also the efficiency of verification. Before describing this methodology, we set below the context in which it will be applied.

3. MODEL CHECKING GROUP AWARENESS IN THE WWW

Group awareness has received attention from the CSCW community for over a decade now, yet the term is not defined uniquely because it spans a broad range of issues. There exist in fact several types of awareness, whose definitions are not mutually exclusive. For the purpose of this paper we have adopted the definition of Dourish and Bellotti [21], which implies that cooperative work can be coordinated by providing feedback on the work context. In other words, group awareness can be realized as information on user activities and presence, which are represented in this paper by concurrent events. By utilizing formal methods, we intend to investigate how the coordination of these events can enhance group awareness in Web-based CSCW.

We consider for this purpose the protocol of Palfreyman and Rodden and the awareness-support protocol of MetaWeb, and we examine in them several properties with the aid of model checking. Although there exist quite a few protocols for awareness support in the WWW (as we mentioned in Section 1.1), the above two protocols are

representative of the main approaches within this context, namely *passive* (query-based) and *active* (notification-based) awareness. The two protocols are described first informally and then in terms of state-transition models and CTL formulae, which are used afterwards to guide the encoding into SMV. The aim of the model checking experiments that follow the encoding is to infer whether the events that indicate user activities are presented to the attention of group members **(i)** *anyway* [21], **(ii)** without significant *delay* [29], **(iii)** in the *right order* [13, 60], as well as whether **(iv)** the presented events are always the *intended* ones [52].

The first represents a fairness requirement, while the second requirement accrues from the fact that the relevance of awareness information depends heavily on time; in fact, if this information is provided belatedly, it may loose its value (in cooperative editing, for example, two co-authors may mistakenly erase a paragraph if they do not inform each other of their actions). The last two requirements, finally, refer to *interface consistency* (as indicated by Sun *et al*. [52]) that we deal with in Section 6.

3.1. The Protocol of Palfreyman and Rodden

3.1.1. Informal Description

This protocol runs atop TCP/IP and supports awareness with a parallel communication facility, whereby collaborators can capture each other's presence and actions. Specifically, each Web site executing this protocol runs two servers, i.e., a regular HTTP server and a server providing awareness information. This information denotes the presence of clients and is recorded in HTML pages. So for example clients can be located with standard CGI programs [27] at a Web site, and be called then either through static links in HTML pages or through HTML Forms. Each Web client has an associated 'awareness client', whose role is to sign onto the server holding the awareness information whenever the Web client requests some document from the HTTP server. Sign-on messages have a parameter specifying the exact URL of the awareness server and they are preceded by the total message length to assist error detection.

Upon receiving a sign-on message, the awareness server adds the client's details to a list of clients who are accessing this URL at the same time. Moving to a new document is only possible if the Web client has instructed first the associated awareness client to sign off from the awareness server. By maintaining the various sign-on and sign-off messages, the awareness server enables clients to obtain information on each other's presence and actions, such as how many of them (and who) are accessing a specific URL, how many pages are accessed concurrently at a particular point in time, etc. This information can be obtained by sending query messages defined especially for this purpose.

The above description defines four types of interaction, which are shown in Fig. (**2a**):

In the following sections, such interactions are specified by state-transition models and CTL formulae.

3.1.2. Specification by a State-Transition Model

The model we have chosen to specify the protocols' behavior in is a special kind of a *finitestate automaton*. In general, finitestate automata are powerful means for representing concurrency [22]. To illustrate their use in the specification of Palfreyman and Rodden's protocol, we consider a simple property that we express first in CTL:

$$\forall\Box((\text{client} = \text{idle}) \rightarrow \forall\Diamond(\text{client} = \text{signs-on})) \quad (1)$$

The above formula means that an idle client will eventually sign on to an awareness server. The concatenation of the *universal* quantifier '$\forall$' and the *henceforth* operator '$\Box$' in the beginning of the formula denotes that the formula will always be True from now on, while the concatenation of the *universal* quantifier and the *eventually* operator '$\Diamond$' after the implication sign denotes that the proposition on the right will hold eventually at some time in the future. The transition implied by the above formula (i.e., that the client will move from the idle to the signs-on state) is depicted in the automaton of Fig. (**3**) by an arrow connecting the corresponding states (i.e., circles).

Because the client is idle initially, the corresponding state is marked in Fig. (**3**) with an arrow without origin. The client may remain in that state for some time, so there is another arrow originating from it and ending back to itself. This state is also a 'final' one (which is denoted by a double circle). Clients and servers are represented by distinct automata, which are composed to represent the interactions implied from the protocol's description. Although in Fig. (**2**) we have drawn two clients, in Fig. (**3**) there is only one automaton representing client behaviour, which is composed with the server's automaton *via* four dotted arrows.

In general, there is a direct correspondence between temporal formulae and finite-state automata [38, 61], but the correspondence between formula (1) and the aforementioned

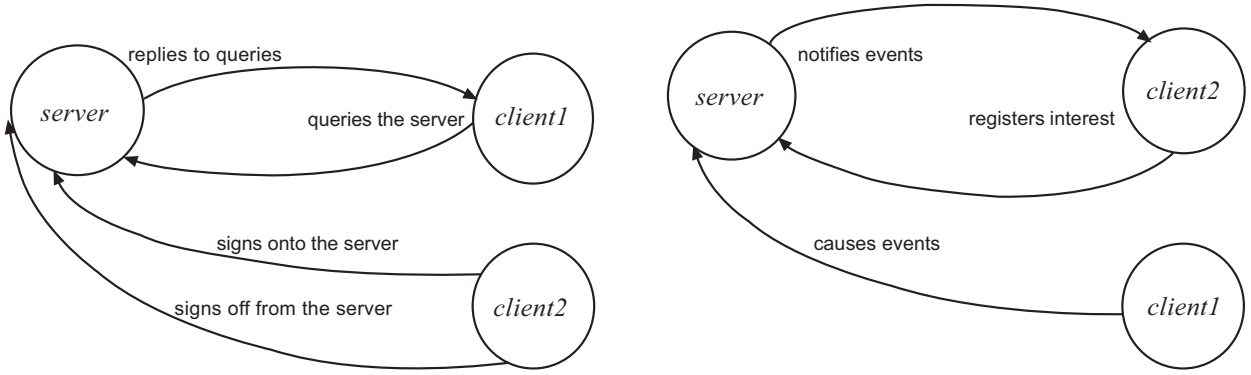

Fig. (2). Possible interactions in the two protocols.

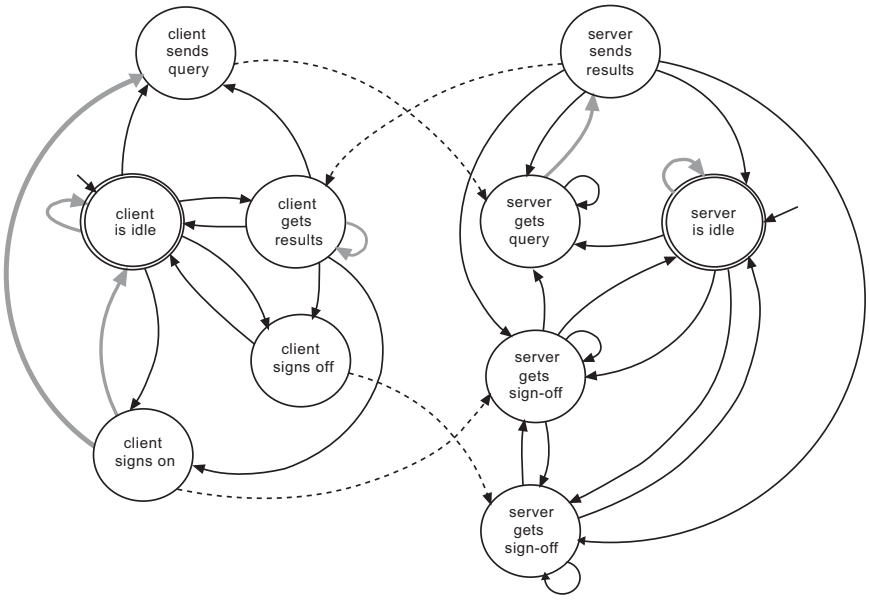

Fig. (3). Finitestate automaton representing Palfreyman and Rodden's protocol.

transition in Fig. (**3**) is not such. In fact, following [38], formula (1) should correspond to the automaton of Fig. (**4**) below, in which there is an arrow from the signs-on state to itself labeled with the null symbol '∅'. In Fig. (**3**) we have omitted this arrow and the null symbol for brevity, and we assume that the transitions are asynchronous. Some transitions which are obviously synchronous though (like the transition from the gets-query to the sends-results state) have been drawn with bold arrows.

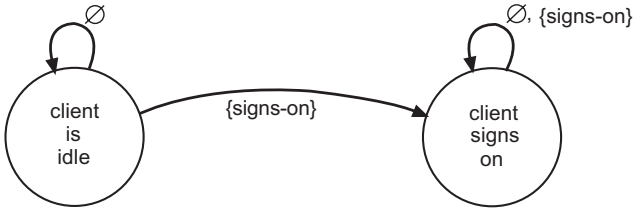

Fig. (4). Finite-state automaton indicating an action that will happen eventually.

Besides the above property, there are also some properties in Palfreyman and Rodden's protocol which denote *dependencies* among events and cannot be represented graphically by a single transition. Formula (2) below, for example, denotes the property that a client cannot submit two queries in sequence without having received in between the results of the first query:

$$\forall\Box((\text{client} = \text{sends-query}) \rightarrow \forall\Diamond(\forall[\neg\ (\text{client} = \text{sends-query})\ U\ (\text{client} = \text{gets-results})])) \quad (2)$$

In the automaton of Fig. (**3**) this property is represented by four transitions, which connect the states client-sends-query, server-gets-query, server-sends-results, client-gets-results, and client-sends-query. The direct encoding of the protocol properties with the methodology we referred to earlier is meant to avoid the task of drawing several transitions for properties like this. This methodology incurs by no means loss of semantic information since, as we mentioned earlier, temporal formulae can be converted into finite-state automata, so the direct encoding of the former into SMV is equivalent to the encoding based on their corresponding automata. The semantic equivalence of our methodology with automata-based encodings is further discussed in Section 4.2, where we argue for its soundness using specific examples.

Below we complete the formal description of Palfreyman and Rodden's protocol, by specifying in CTL some other properties[1] that we deem essential.

3.1.3. Protocol Specification in CTL

$$\forall\Box((\text{client} = \text{signs-on}) \rightarrow \forall\Diamond(\text{client} = \text{signs-off})) \quad (3)$$

The above formula means that if a client has signed on to a URL, after some time s/he will eventually sign off. The property expressed by this formula is represented graphically in Fig. (**3**) by two transitions between the signs-on and the idle states and between the idle and the signs-off states, respectively. We have not drawn a loop transition from the signs-on state to itself, because this would contradict the requirement that a client may not sign onto a new URL if s/he has not signed off from the previous one. This is in fact a *safety* requirement [40] that can be expressed in CTL by the following formula (which extends formula (3)):

$$\forall\Box((\text{client} = \text{signs-on}) \rightarrow \forall\Diamond(\forall[\neg\ (\text{client} = \text{signs-on})\ U\ (\text{client} = \text{signs-off})])) \quad (4)$$

Similarly to formula (3), the formula below denotes the intention of a client to sign off from the current URL in order to move to another one:

$$\forall\Box((\text{client} = \text{signs-off}) \rightarrow \forall\Diamond(\text{client} = \text{signs-on})) \quad (5)$$

This property is represented in Fig. (**3**) by two transitions, which connect the signs-off, the idle, and the signs-on states. The transition that connects the first two of these states denotes also that a client may leave a session after signing off from the awareness server.

[1]The protocol has been implemented in C++ and its source code was until recently available from the Web site of Lancaster University (Computing Department). The specification of the protocol's properties is based on that code as well as on the informal description above.

$$\forall\Box((\text{server} = \text{gets-query}) \rightarrow \forall\bigcirc(\text{server} = \text{sends-results})) \quad (6)$$

i.e., upon getting a query from a client, the server releases the results immediately.

$$\forall\Box((\text{client} = \text{gets-results}) \rightarrow \forall\bigcirc((\text{client} = \text{signs-on}) \vee (\text{client} = \text{signs-off}) \vee (\text{client} = \text{idle}) \vee$$

$$(\text{client} = \text{gets-results}) \vee (\text{client} = \text{sends-query})) \quad (7)$$

The above formula denotes that, upon receiving the results of a query, a client may leap to any other state afterwards. In fact, in Fig. (**3**) this property is denoted by direct transitions from the gets-results state to every other state of the client's automaton. The loop transition from that state to itself denotes that a client may keep receiving results for a short while (due to network latency).

$$\forall\Box((\text{client} = \text{signs-on}) \rightarrow \forall\bigcirc((\text{client} = \text{idle}) \vee (\text{client} = \text{sends-query})) \quad (8)$$

Upon signing on to a URL, a client may remain idle or submit a query. Yet s/he may not get the results of a previous query, since this must happen before s/he signs on to a URL.

$$\forall\Box((\text{client} = \text{signs-on}) \rightarrow \forall\Diamond(\text{server} = \text{gets-sign-on})) \quad (9)$$

In the above formula, the concatenation of the *universal* quantifier and the *eventually* operator denotes that the server may not get immediately the sign-on message, due to network latency. The same holds for the four formulae below:

$$\forall\Box((\text{client} = \text{signs-off}) \rightarrow \forall\Diamond(\text{server} = \text{gets-sign-off})) \quad (10)$$

$$\forall\Box((\text{client} = \text{sends-query}) \rightarrow \forall\Diamond(\text{server} = \text{gets-query})) \quad (11)$$

$$\forall\Box((\text{server} = \text{sends-results}) \rightarrow \forall\Diamond(\text{client} = \text{gets- results})) \quad (12)$$

$$\forall\Box((\text{client} = \text{idle}) \rightarrow \forall\Diamond((\text{client} = \text{idle}) \vee (\text{client} = \text{sends-query}) \vee (\text{client} = \text{gets-results}) \vee$$

$$(\text{client} = \text{signs-on}) \vee (\text{client} = \text{signs-off}))) \quad (13)$$

Formula (13) generalizes formula (1) and the property it denotes is represented in Fig. (**3**) by the various transitions that connect the idle state with all other states in the client's automaton.

Formula (14), in turn, denotes what a server may do after the receipt of a sign-on message from a client:

$$\forall\Box((\text{server} = \text{gets-sign-on}) \rightarrow (\forall\Diamond((\text{server} = \text{gets-sign-off}) \vee (\text{server} = \text{gets-sign-on}) \vee$$

$$(\text{server} = \text{gets-query}))) \vee \forall\bigcirc(\text{server} = \text{idle}) \quad (14)$$

Formula (15), on the other hand, denotes the actions of the server upon receiving a sign-off message:

$$\forall\Box((\text{server} = \text{gets-sign-off}) \rightarrow (\forall\Diamond((\text{server} = \text{gets-sign-off}) \vee (\text{server} = \text{gets-sign-on}))) \vee$$

$$\forall\bigcirc(\text{server} = \text{idle}) \quad (15)$$

Formula (16), finally, denotes the possible actions of a server when it is in the idle state:

$$\forall\Box((\text{server} = \text{idle}) \rightarrow \forall\Diamond((\text{server} = \text{idle}) \vee (\text{server} = \text{gets-query}) \vee$$

$$(\text{server} = \text{gets-sign-on}) \vee (\text{server} = \text{gets-signs-off}))) \quad (16)$$

The above formally expressed properties form a representative picture of Palfreyman and Rodden's protocol, and they are encoded later into SMV based on the methodology we have in-vented. Below we describe the awareness-support protocol of MetaWeb, which is also encoded with this methodology later.

3.2. Awareness Support in MetaWeb

3.2.1. Informal Description

The primary concern behind the development of MetaWeb was the insufficiency of other groupware systems at that time to provide synchronous session coupling to Web pages. Another concern was platform and browser independence, while the need for supporting diverse groupware applications across the WWW was also taken into account.

MetaWeb couples Web pages to collaborative sessions *via* access lists, on which it registers client interests and maintains active sessions. Interest registration is done once for each client when s/he joins a session. Clients and servers exchange events which are triggered whenever a client moves to a new page, modifies it, or joins a new session on the same page. These events are defined as Java objects, including among their attributes a sequence number for message ordering and concurrency control. Event notification is always done explicitly and it depends on whether a client has registered an interest in all the events of a session or in specific ones only. In both cases, upon an event's occurrence the server is informed accordingly and notifies then that event to all the clients who have registered a relevant interest. The various interactions between the components of this protocol are depicted in the producer-consumer model of Fig. (**2b**). Moreover, they can be represented by a finite-state model, as in Fig. (**5**) below:

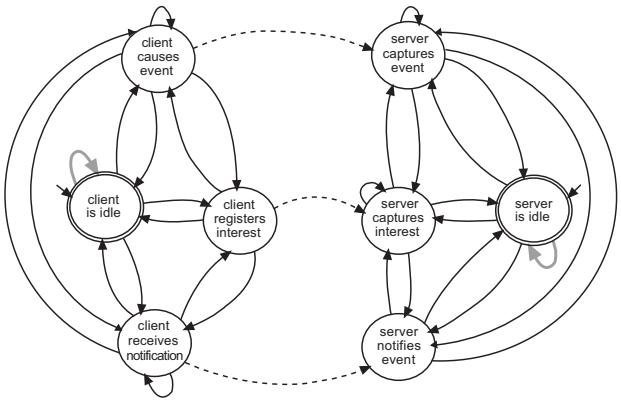

Fig. (5). Finitestate automaton for the Awareness-support Protocol of MetaWeb.

There are again two distinct automata in this figure (i.e., one for clients and one for servers), which are composed *via* three dotted arrows. The actual encoding of MetaWeb was derived from CTL formulae though (as was the encoding of Palfreyman and Rodden's protocol). Some of these formulae are presented later in this paper in order to assist the interpretation of the model checking results.

4. A NEW ENCODING METHODOLOGY FOR SMV

Having outlined so far the functionality of the two protocols with temporal properties and finitestate models, we present in the current section a novel methodology for transforming CTL formulae into SMV code, which will assist us subsequently in the compact representation of the protocols' behavior in this model checker and the verification of key properties of awareness support. Our methodology comprises seven rules that are empirical but, as we show later, they reduce substantially the encoding effort and simplify verification. Regarding the novelty of the methodology, it is discussed in Section 4.2 below.

4.1. Encoding Rules

Rule I. Implications followed by the *universal* quantifier and the *next* operator are encoded as deterministic next-state assignments. For example, the formula $\forall\Box(p(v_1) \rightarrow \forall\bigcirc q(v_2))$ is encoded as

next(v_2) :=
  case
    $p(v_1)$: $q(v_2)$;
    1 : <current-value>;
  esac;

In this example, $p(v_1)$ represents a proposition involving the system variable v_1 (e.g., $p(v_1) \equiv (v_1 = gets\text{-}message)$). Similarly, $q(v_2)$ represents another proposition involving the system variable v_2.

Rule II. Implications followed by the concatenation '$\exists\bigcirc$' are encoded as non-deterministic assignments that include the proposition in the right part of the formula (referred to henceforth as 'implied proposition') and some *other* value, which must have been declared as a symbolic-type variable in the VAR section of the encoding. For example, the formula $\forall\Box(p(v_1) \rightarrow \exists\bigcirc q(v_2))$ is encoded as

next(v_2) :=
  case
    $p(v_1)$: {$q(v_2)$, *other*};
    1 : <current-value>;
  esac;

Rule III. Concerning variables that represent system entities not acting on their own (like the server variable in Palfreyman and Rodden's protocol), their default next-state value is the same as the current.

Rule IV. Implications denoting eventuality through the concatenation '$\forall\Diamond$' are encoded as non-deterministic assignments, which include the implied proposition of the formula, the *in-transit* state (which must be declared in the VAR section), as well as a counter variable whose type is an integer subrange. For example, the formula $\forall\Box(p(v_1) \rightarrow \forall\Diamond q(v_2))$ is encoded as

next(v_2) :=
  case
    (counter = 0) & $p(v_1)$: *in-transit*;
    (v_2 = *in-transit*) & (counter < MAX) : {$q(v_2)$, *in-transit*};
    (v_2 = *in-transit*) & (counter = MAX): $q(v_2)$;
    1 : <current-value>;
  esac;

Obviously, the integer subrange of counter is 0 . . . MAX. The selection of a subrange for counter is necessary, because we do not know when the proposition $q(v_2)$ starts to hold so we assume a certain time interval (i.e., (0, MAX]) within which this can happen. Otherwise, the non-deterministic assignments might force SMV to enter a time-consuming loop and cause thereby state explosion.

To complete the encoding of the above formula we must perform next-state evaluations for the counter variable too, since the value of this variable changes in each transition and $q(v_2)$ depends directly on that change. We encode counter as follows:

init(counter) := 0;
next(counter) :=
  case
    (v_2 = *in-transit*) : counter + 1;
    1 : counter;
  esac;

If in the module that contains the encoding of $\forall\Box(p(v_1) \rightarrow \forall\Diamond q(v_2))$ there is also another encoded formula, in which $p(v_1)$ is implied by another proposition, the next-state values of counter must be computed as follows:

next(counter) :=
  case
    $p(v_1)$: $counter_0 + 1$;
    (v_2 = *in-transit*) : counter + 1;
    1 : $counter_0$;
  esac;

where $counter_0$ is the value of the counter resulted from the evaluation of the other formula. The next-state evaluation of v_2 must be rewritten in that case as follows:

next(v_2) :=
  case
    (counter = $counter_0$) & $p(v_1)$: *in-transit*$_1$;
    (v_2 = *in-transit*$_1$) & (counter < MAX +
    $counter_0$) : {$q(v_2)$, *in-transit*$_1$};
    (v_2 = *in-transit*$_1$) & (counter = MAX +
    $counter_0$) : $q(v_2)$;
    1 : <current-value>;
  esac;

The above rule can be used to encode also the formula $\forall\Box(p(v_1) \rightarrow \forall\bigcirc q(v_2))$, but in the first command of the next-state assignment we must add an *other* value (i.e., write (counter = 0) & $p(v_1)$: {*in-transit*, *other*};)

Rule V. Formulae containing the *until* operator after an implication sign can be encoded in several ways.

Specifically, the formula $\forall\Box(p(v_1) \rightarrow \forall[q(v_2)\ \mathrm{U}\ r(v_2)])$ is encoded by the assignment

next(v_2) :=
case
$p(v_1)$ & (counter$_1$ = 0) : $q(v_2)$;
$q(v_2)$ & (counter$_1$ < MAX$_1$) : {$q(v_2)$, $r(v_2)$};
$q(v_2)$ & (counter$_1$ = MAX$_1$) : $r(v_2)$;
1 : <current-value>;
esac;

Here we also assume a time interval within which $r(v_3)$ may start holding (and thus invalidate $q(v_2)$), so counter$_1$ must be evaluated the same way as counter. Depending on the meaning of the above formula in a real domain, the value of MAX$_1$ may be set higher or lower than the value of MAX in the previous rule.

Concerning in turn the formula $\forall\Box(p(v_1) \rightarrow \forall[\neg\, q(v_2)\ \mathrm{U}\ r(v_2)])$, it is encoded similarly except that we must assume one or more values of v_2 that imply the negation of $q(v_2)$. For example, we can declare in the beginning of the encoding

VAR
v_2 : { y_1, y_2, y_3, y_4, y_5};

(where $v_2 = y_1$ implies the satisfaction of $q(v_2)$, { $v_2 = y_i$, $2 \le i \le 4$ } imply its negation, and $v_2 = y_5$ implies the satisfaction of $r(v_2)$) and then encode the above formula as follows:

next(v_2) :=
case
$p(v_1)$ & (counter$_2$ = 0) : {y_2, y_3, y_4};
($v_2 = y_2 \mid v_2 = y_3 \mid v_2 = y_4$) & (counter$_2$ < MAX$_2$) : {y_2, y_3, y_4, y_5};
($v_2 = y_2 \mid v_2 = y_3 \mid v_2 = y_4$) & (counter$_2$ = MAX$_2$) : y_5;
1 : <current-value>;
esac;

Combining the above with Rule IV, we can encode the formula $\forall\Box(p(v_1) \rightarrow \forall\Diamond(\forall[\neg q(v_2)\ \mathrm{U}\ r(v_2)]))$ as follows:

next(v_2) :=
case
$p(v_1)$ & (counter$_3$ = 0) : *in-transit*$_2$;
(v_2 = *in-transit*$_2$) & (counter$_3$ < MAX$_3$) & (counter$_4$ = 0) : {*in-transit*$_2$, y_2, y_3, y_4};
(v_2 = *in-transit*$_2$) & (counter$_3$ = MAX$_3$) & (counter$_4$ = 0) : {y_2, y_3, y_4};
($v_2 = y_2 \mid v_2 = y_3 \mid v_2 = y_4$) & (counter$_4$ < MAX$_4$) : {y_2, y_3, y_4, y_5};
($v_2 = y_2 \mid v_2 = y_3 \mid v_2 = y_4$) & (counter$_4$ = MAX$_4$) : y_5;
1 : <current-value>;
esac;

Note that in the above encoding the variable counter$_4$ can only increase its value if counter$_3$ has reached its upper limit. Hence counter$_4$ is encoded as

init(counter$_4$) := 0;
next(counter$_4$) :=
case
($v_2 = y_2 \mid v_2 = y_3 \mid v_2 = y_4$) & (counter$_3$ = MAX$_3$) : counter$_4$ + 1;
1 : counter$_4$;
esac;

Concerning finally the formula $\forall\Box(p(v_1) \rightarrow \exists[\ q(v_2)\ \mathrm{U}\ r(v_2)])$, it is encoded as follows:

next(v_2) :=
case
$p(v_1)$ & (counter$_5$ = 0) : {$q(v_2)$, *other*};
$q(v_2)$ & (counter$_5$ < MAX$_5$) : {$q(v_2)$, $r(v_2)$};
$q(v_2)$ & (counter$_5$ = MAX$_5$) : $r(v_2)$;
1 : <current-value>;
esac;

Rule VI. Formulae containing the *releases* operator after an implication sign are encoded similarly, since this operator is the dual of *until* (i.e., $\forall[q(v_2)\ \mathrm{U}\ r(v_2)] \equiv \forall[r(v_2)\ \mathrm{R}\ q(v_2)]$).

Rule VII. A CTL formula with two implication signs (like $p \rightarrow (q \rightarrow r)$ for example) is encoded as follows:

If the second implication sign is followed by the concatenation '$\forall\bigcirc$', i.e., $\forall\Box(p(v_1) \rightarrow (q(v_2) \rightarrow \forall\bigcirc r(v_3)))$

(where v_1, v_2, and v_3 represent system variables and $v_1 \neq v_2$), then the formula is encoded as a deterministic next-state assignment:

$$\mathrm{next}(v_3) := p(v_1)\ \&\ q(v_2) : r(v_3);$$

The above assignment follows the tautology $p \rightarrow (q \rightarrow r) \equiv (p \wedge q) \rightarrow r$ of propositional logic.

Instead, if the second implication sign is followed by the concatenation '$\forall\Diamond$', (e.g., $\forall\Box(p(v_1) \rightarrow (q(v_2) \rightarrow \forall\Diamond r(v_3)))$), then the corresponding assignment will indicate a non-deterministic choice, which must be associated with a counter variable as before:

next(v_3) :=
case
$p(v_1)$ & $q(v_2)$ & (counter$_6$ = 0) : *in-transit*$_3$;
(v_3 = *in-transit*$_3$) & (counter$_6$ < MAX$_6$) : {$r(v_3)$, *in-transit*$_3$};
(v_3 = *in-transit*$_3$) & (counter$_6$ = MAX$_6$) : $r(v_3)$;
1 : <current-value>;
esac;

Finally, in the case where v_1 and v_2 correspond to the same system variable, they must be encoded in SMV by two distinct variables.

4.2. Discussion of the Methodology

The main innovation introduced by the above methodology is the use of *counter* variables, which are meant to control the recursive execution of some assignment

statements. In fact, the recursive execution of non-deterministic assignments (like the encoding of the '∀◊' pair) contributes to state explosion and can make model checking impractical, despite the remarkable efficiency of SMV. By confining the counter variables within a narrow subrange we impose a restriction on this execution, so the latter is *guaranteed* to terminate within a finite number of steps (this is also the prespective of SAT-based symbolic model checking that we mentioned in Section 2.3, which is sometimes referred to in the literature as *bounded model checking* [19]). In the encoding of the two protocols, for example, we have assumed an integer subrange between 1 and 1000, which means that the granularity of transitions ranges between 1 msec and 1 sec.

In fact, as the awareness messages that are exchanged among group participants are small in size, the total transmission time is dominated by network latency. According to [53], the latency on the Internet rarely exceeds 1 sec, so awareness messages are normally delivered within that slot. By assigning therefore the subrange 1 . . 1000 to control variables we implicitly assume that at most 1000 transitions can take place in the state space (each one corresponding to 1 msec) whenever two entities exchange messages. This way we restrict the size of that space and reduce the possibility of state explosion. Although bounded model checking is also meant to increase efficiency, there are two fundamental differences between it and our encoding methodology, i.e., (1^{st}) bounded model checking does not restrict the size of state space (as our methodology does) and (2^{nd}) our methodology does not only assist in error detection (as it is usually the case with bounded model checking), but it can also prove the absence of errors in groupware protocols by taking advantage of the aforementioned bound on Internet latency. Hence our methodology is unique.

Although SMV is inefficient in manipulating integers, the subrange of 1 to 1000 is so narrow that it cannot cause a state explosion. In other cases, however, this subrange may not be appropriate (e.g., in cooperative transaction processing or virtual environments); to avoid a state explosion in those cases, one may convert integer variables into bit strings, as suggested by Chan *et al.* [14, p.510].

The methodology is also complete in the sense that all the temporal operators (and their possible combinations with path quantifiers) are encoded, each one with a different rule. Regarding in turn the soundness of the methodology, it depends on whether the encoding rules can represent the behavior of a system as adequately as ordinary encoding methods that rely on state-transition models. As we mentioned earlier, all temporal operators and their combinations with path quantifiers can be easily converted into finite-state automata, so their respective encodings into SMV must be the same. The only case where our methodology differs significantly from automata-based encodings is when multiple transitions of an automaton are represented in our methodology by a single formula, which does not contain some of the states accessed by those transitions. The property denoted by formula (2), for example, corresponds in Fig. (**3**) to four transitions, while two of the states accessed by those transitions (i.e., gets-query and sends-results) do not appear in formula (2) (either explicitly or implicitly). Hence the respective encodings will be different.

We have also pointed out earlier that the encoding effort is reduced if we encode formula (2) directly into SMV. In fact, it is preferable to write a single CTL formula in one line than drawing four distinct states and connecting them with arrows. The SMV code corresponding to that formula is 15 lines totally (see Section A.5 of the Appendix), i.e., 5 lines for the formula itself and 5 for each counter variable, whereas the code corresponding to the four transitions of Fig. (**3**) is larger since three of them are asynchronous (and thus they cannot be encoded with a single assignment that requires 4 lines of code). Yet in the protocol specification we have also included CTL formulae for the properties denoted by these transitions, so the overall gain in terms of effort is moderated. On the other hand though, we must note that we obtain additional semantic information from the encoding of temporal formulae into SMV. For example, if we attempt to verify a property that contradicts formula (2), the encoding of the latter will assist the model checker to produce counterexamples at a faster pace than the encoding of the four transitions of Fig. (**3**).

In summary, we can say that the benefits of our encoding methodology can be fully exploited if developers specify *adequately* a system in CTL, so as to capture every aspect of its behavior. This holds nevertheless for transition-based encodings as well, since a state-transition model that has not captured adequately the behavior of a system is useless to model checking.

5. THE METHODOLOGY IN USE

5.1. Encoding the Protocol of Palfreyman and Rodden

This protocol was illustrated in Fig. (**2a**) by a fairly abstract model, which conveyed the philosophy that awareness support should be regarded as a centralized communication process. Based on this model, we specified in Section 3.1.2 a number of properties.

In the current section, we show how these properties guided the protocol's encoding into SMV based on the methodology we presented above. A complete encoding example is presented in Section A.5 of the Appendix, where we have assumed for simplicity only one client. In the model checking experiments, however, we included several client submodules (i.e., processes) that were allowed to run concurrently. The correctness properties examined in these experiments are described below.

5.1.1. Encoding Correctness Properties

Besides assignment statements, the encoding of Palfreyman and Rodden's protocol includes also one SPEC statement. This specification represents the encoding of a property which does not emerge directly from the protocol's description, but it is important because it entails the satisfaction of the first two requirements in the beginning of Section 3. Specifically, this property concerns whether it is possible for a client to remain unaware of other clients' activities (in other words, we are interested in finding whether a client may request information on ongoing or past activities and not receive it). We express this property with the following CTL formula:

$\exists\Diamond$(client = sends-query $\rightarrow$ $\forall\Box$(server-does-not-return-requested-results)) $\equiv$

$\exists\Diamond$(client = sends-query $\rightarrow$ $\forall\Box$($\neg$ server-returns-requested-results))

The proposition server-returns-requested-results can be True if and only if

(i) the server always replies to client requests and

(ii) any information requested by the clients is always available.

Since client activities are temporally delimited by sign-on and sign-off messages, the availability of awareness information can be denoted by the proposition a-signed-on-client-has-not-signed-off-yet. Hence the CTL formula above is equivalent to

$\exists\Diamond$(client = sends-query $\rightarrow$ $\forall\Box\neg$(server = sends-results

a-signed-on-client-has-not-signed-off-yet)) $\equiv$ $\exists\Diamond$(client = sends-query $\rightarrow$

$\forall\Box$($\neg$ (server = sends-results) ($\neg$ a-signed-on-client-has-not-signed-off-yet))) $\equiv$

$\exists\Diamond$(client = sends-query $\rightarrow$ ($\forall\Box\neg$ (server = sends-results)

($\forall\Box\neg$ a-signed-on-client-has-not-signed-off-yet))) $\equiv$

$\exists\Diamond$(client = sends-query $\rightarrow$ $\forall\Box\neg$ (server = sends-results))

$\exists\Diamond$(client = sends-query $\rightarrow$ $\forall\Box\neg$ a-signed-on-client-has-not-signed-off-yet)

The first implication in the above disjunction is False, as implied by formulae (11) and (6) in Section 3.1.3. Thus the validity of the disjunction depends on the second implication, i.e.,

$\exists\Diamond$(client = sends-query $\rightarrow$ $\forall\Box\neg$ a-signed-on-client-has-not-signed-off-yet) $\equiv$

$\exists\Diamond$((client = sends-query) $\rightarrow$ $\forall\Box$ all-signed-on-clients-have-signed-off)

Still CTL cannot express predicates like this, because it does not contain past-time operators. The meaning of the above formula can be expressed however by the equivalent one

$\exists\Diamond$((client = signs-on) $\rightarrow$ $\forall$[$\neg$ (client = sends-query) U (client = signs-off)])
(17)

Formula (17) denotes that while a client is accessing various URLs, another client does not send queries so s/he remains unaware of the first client's activities. This formula expresses accurately the intended meaning, but before encoding it we inverted that meaning. The rationale behind this was that we wanted to derive counterexamples so as to identify specific cases of faulty behavior. Had we encoded the original formula as it was, the model checker would confirm that a client can remain unaware of the activities of other clients without specifying though how this can happen.

Another property we were also interested in was whether information delivery on one's activities can be delayed for some reason. This indeed can happen in two cases:

(i) When the delivery of sign-on and sign-off messages or the server's response to client queries are not instantaneous.

(ii) When a client is informed on another client's activity much later than the occurrence of that activity.

The first case is possible, in fact, as is implied by formulae (9), (10), and (11) in Section 3.1.3. Since its negation would contradict these formulae, we did not incorporate any relevant statement in the SPEC section of the encoding. Concerning case **(ii)**, it can be specified with the following formula:

$\exists\Diamond$((client = signs-on) $\rightarrow$ $\forall\bigcirc$($\neg$ (client = gets-results)))

In fact, the commencement of a client's activity is signified by a sign-on message, which can be captured by other clients through the submission of relevant queries. The fastest way that this can be done is when the server replies to a client's query immediately after the receipt of a sign-on message from another client. As we mentioned earlier, however, the reply of the server cannot reach immediately the interested client, so this property was not incorporated in the encoding.

Concerning finally the requirements about interface consistency, we did not address them because interface consistency is only possible with active notifications, which are not allowed by Palfreyman and Rodden's protocol. As we show in Section 5.3, these requirements are not fully satisfied by MetaWeb either.

5.2. Encoding the Awareness-support Protocol of MetaWeb

The abstract model of group awareness in MetaWeb (Fig. (**2b**)) comprises two clients and one server. The first client acts as a producer of events and the second as a consumer. Similarly to the encoding of Palfreyman and Rodden's protocol, in the encoding of MetaWeb the clients and the server were represented by distinct variables. At the end of this encoding there were three specifications, which were derived similarly to the specification in Palfreyman and Rodden's protocol. For example, in order to check whether a client can remain unaware of other clients' activities or a client's activity can be notified belatedly, we proceeded as follows:

Since awareness is supported by active notifications, the first case is only possible when a client's event is never notified to other clients who have registered a relevant interest. This can be expressed as

$\exists\Diamond$(client = causes-event $\rightarrow$ $\forall\Box$($\neg$ (client = receives-notification))) (18)

Concerning in turn delayed notification, it may happen if a client's event is not captured immediately by the server or is not notified immediately to other interested clients. These possibilities can be expressed by the following formulae:

$\exists\Diamond$(client = causes-event $\rightarrow$ $\forall\bigcirc$($\neg$ (server = captures-event))) (19)

$\exists\Diamond$(client = causes-event $\rightarrow$ $\forall\bigcirc$($\neg$ (client = receives-notification))) (20)

As before, we negated the meaning of the above formulae before encoding them into SMV. Concerning finally the requirements about interface consistency, they are examined within a broader framework in Section 6.2.

5.3. Model Checking Results

Our model checking experiments were performed with the 2.5.0 version of SMV on a Pentium III machine, which had 512 MBytes of RAM and was running Linux. The size of the state space of Palfreyman and Rodden's protocol was 1.1×10^{38}, while that of the state space of MetaWeb was 4.3×10^{37}. Three of the SPEC statements in the encodings of these protocols were checked within seconds or minutes, but some counterexamples took over an hour. Bearing in mind though the results of similar experiments in the literature (e.g., [14], [48], etc), the above results indicate a good performance.

Based on this methodology we wrote and run two scenarios for each protocol, one involving five clients and another involving ten. So each scenario included a main module and five or ten submodule instances that were allowed to run simultaneously. We assumed one server variable in each scenario, but the client variable of each submodule was named differently, e.g., client1, client2, etc. We also assigned different values to these variables, e.g., the various *in-transit* values for client1 were named differently from the corresponding values for client2, and so on. Throughout the experiments we assumed that each client could act both as a producer and consumer of awareness information, which is usually the case in cooperative work. Due to the existence of non-deterministic assignments in the encodings, SMV produced several counterexamples on which we comment below. The presentation of these counterexamples is concise, since only the values of the client and the server make sense, while the values of the counter variables are not of immediate interest.

5.3.1. Model Checking Results for the Protocol of Palfreyman and Rodden

SMV invalidated the SPEC statement in the protocol's encoding. This entails that the requirements for fairness and interactive responsiveness expressed in the beginning of Section 3 are sometimes not satisfied by this protocol. The counterexamples generated by the model checker were presented as several steps in sequence. One of these counterexamples is shown below:

– – specification AG(client2 = signs-on –> E [. . . is false

– – as demonstrated by the following execution sequence

state 1.1 : client1 = idle
client2 = idle
server = idle

state 1.2 : [executing process p1]
client2 = signs-on

state 1.3 : client2 = not-signs-on

state 1.4 : client2 = signs-off
server = gets-sign-on

state 1.5 : [executing process p2]
client1 = sends-query
client2 = in-tr32
server = idle

state 1.6 : client1 = in-tr41
server = gets-sign-off

state 1.7 : [executing process p1]
server = idle

state 1.8 : server = gets-query

As we mentioned earlier, the encoding of the second scenario included ten instances of a sub-module comprising the client and its associated counter variables, as well as their next-state evaluations. Variables p1 and p2 in the above counterexample represent two such instances. This counterexample demonstrates that if a client queries the awareness server while another client is signing off, the query will not produce any results on the activities of the second client. SMV generated several other counterexamples, in one of which client2 was shown to query the awareness server before client1 had signed on but not submitting any other queries afterwards.

5.3.2. Model Checking Results for MetaWeb

MetaWeb's model checking examined the negations of the three CTL formulae in Section 5.2. SMV generated several counterexamples, demonstrating thus the protocol's inability to guarantee fairness and preserve interface consistency. The lack of fairness, in particular, was demonstrated by the following counterexample which includes (among others) an *in-transit* value that had been declared in the encoding:

– – specification AG(client1 = causes-event –> AF client2 = receives-notification) is false

– – as demonstrated by the following execution sequence

state 1.1 : (. . . initializing system variables . . .)

state 1.2 : client1 = causes-event

state 1.3 : client1 = in-tr11

state 1.4 : [executing process p1]
client1 = idle
server = captures-event

state 1.5 : [executing process p2]
client2 = registers-interest
server = idle

This counterexample shows that MetaWeb cannot provide awareness information on past activities (i.e., if a client registers her/his interest in a certain event after the occurrence of that event, s/he will never receive a relevant notification). When we run the scenario with ten clients, SMV produced several variations of the above counterexample, including for example the generation of multiple events simultaneously with the registration of multiple client interests (owing to the fact that we had allowed the interleaved execution of submodules). This counterexample falsified also the third specification, while the second specification was falsified because it is impossible to achieve immediate information delivery (as denoted by the *next* operator) in an asynchronous environment like the WWW.

6. DISCUSSION

6.1. General Comments

Palfreyman and Rodden's protocol provides a good degree of presence awareness but does not provide sufficient activity awareness in some cases. As demonstrated by the counterexample in Section 5.3.1, if a client signs onto a URL while another client is performing some action and then the first client signs off from that URL before the second client completes his action, the second client will remain unaware of that action.

MetaWeb avoids the above problem thanks to the active notification of events upon their occurrence. Yet awareness support may still be insufficient, due to network latency and the time needed to establish HTTP links. So while an event is along the way to a client, the latter may be doing several actions which, upon the event's arrival, will need to be *undone* [1]. If these actions have triggered real-world activities in the meantime though, their undo will not have the desired effects.

Besides the sensitivity to network latency, model checking demonstrated that MetaWeb prohibits clients to receive feedback on past activities. However this feedback is required quite often, so several CSCW systems tend to record *interaction histories* for this purpose (e.g., the framework presented in [5]). The above insufficiency of MetaWeb owes to the fact that its awareness-support protocol disposes events if no client has registered an interest in them. The protocol of Palfreyman and Rodden also falls short in this aspect, since it cannot convey information on a client's activities once this client has signed off from a URL.

6.2. Proposed Improvements

6.2.1. Notifying Past Activities

One way to satisfy the requirement for past-activity awareness is to allow some form of *event buffering*. That is, once a client's event is passed to the server, the latter should either notify it immediately to other clients or save it in a buffer for later dispatching (i.e., upon recording a relevant interest).

Event buffering can be specified with the following formulae:

$$\forall\Box((\text{server} = \text{captures-event}) \rightarrow \forall\Diamond(\text{server} = \text{saves-event})) \quad (21)$$

$$\forall\Box((\text{server} = \text{saves-event}) \rightarrow \forall\Diamond((\text{server} = \text{captures-interest}) \rightarrow \forall\bigcirc(\text{server} = \text{notifies-event}))) \quad (22)$$

So in general, the active notification of events (as it is done by MetaWeb) and their temporary buffering (as it is suggested above) are necessary features for awareness support in the WWW. Yet active notification can only guarantee interface consistency under the assumption of immediate message delivery. In the WWW, however, this assumption may not be realistic due to the latency over the Internet.

6.2.2. Model Checking Interface Consistency

We considered a MetaWeb scenario involving more than two clients, who were generating events asynchronously. To distinguish among those events, we changed the definition of the client variable by incorporating multiple instances of its values. The requirement for event notification in the right order was expressed then as

$$\forall\Box(\text{client1} = \text{causes-event1} \rightarrow \forall\Diamond(\text{client1} = \text{causes-event2} \rightarrow \forall\Diamond(\forall[\neg(\text{client2} = \text{receives-notification2}) \text{ U } (\text{client2} = \text{receives-notification1})]))) \quad (23)$$

When we tested this formula with SMV we received a number of counterexamples, which denoted that, due to network latency, the server may notify a recent event earlier than an old one. In cooperative applications, however, the belated arrival of an event may prompt clients to undo several actions. Problems like this can be avoided by associating each event with a *timestamp* denoting the exact time of its generation. Timestamps are more effective than the sequence numbers assigned to events by MetaWeb, since two events of the same client may have the same sequence number (if they belong to different messages) but may be notified to another client simultaneously, due to variations in network latency.

In turn, if the actions taken upon the arrival of out-of-order notifications have triggered other events in the meantime, complete chaos will prevail. This chaos can be detected by timestamps but may be difficult to resolve. Hence the active notification of events cannot guarantee by itself interface consistency.

6.2.3. Refining the Protocol Specification to allow for Interface Consistency

In an attempt to ensure interface consistency, we investigated first the implications of out-of-order notifications in situations like the above. To this end, we refined MetaWeb's encoding by allowing clients to create events and register interests in an interleaving fashion, whereas in the original encoding we had assumed distinct roles for each client. We also added several assignment statements in order to represent causal dependencies between events. For example, using multiple instances of the client variable, we encoded the formulae

$$\forall\Box((\text{client2} = \text{receives-notification2}) \rightarrow \forall\Diamond(\text{client2} = \text{causes-event2})) \quad (24)$$

and

$$\forall\Box((\text{client2} = \text{receives-notification2}) \rightarrow \forall\Diamond(\forall[\neg(\text{client1} = \text{causes-event3}) \text{ U } (\text{client2} = \text{causes-event2})])) \quad (25)$$

which denote that the notification of an event forces the recipient to create another event, without which the first client cannot continue his work. In a cooperative design session, for example, a designer may mark an artifact for deletion and wait then for the approval of the other designers in the group. If the latter are not informed immediately of the former's intention, the deletion will be postponed to the detriment of the design activity.

The counterexamples generated by SMV to the negation of formula (25) were so many, that it would not be possible to capture them with manual methods. These counterexamples indicated situations where

(i) the clients were receiving notifications out of order, which in turn were "freezing" other clients for extended time periods or

(ii) the clients were involved in livelocks by exchanging events irrespective of the causal dependencies between them.

These situations denote lack of scheduling, which owes to the inability of clients to estimate temporally the arrival of notifications. Because user activities are rarely scheduled [51], however, an alternative to the requirement for orderly notifications would be to provide users with feedback on the activity history of their team, so as to allow them to adjust their own activities to the activities of the team. This is a critical usability issue, since users can only arrange their activities from the information presented in their GUI. In the above design session, for example, designers might avoid livelocks if they knew that their colleagues have taken some actions in the meantime (according to Francez [25], the avoidance of livelock indicates *weak fairness*, since actions are offered continuously a chance until the moment they are enacted following the notification of earlier actions in their causal order). Similarly, designers might not get stuck if they knew that the activities of their colleagues had been notified in the wrong order. This can be expressed in general terms as

$\forall\Box$((client = receives-notification) $\rightarrow$ $\forall\Diamond$(client-gets-feedback $\rightarrow \forall\bigcirc$(client-causes-another-event)))

The above formula denotes that after the notification of an event, clients keep interacting with each other on the basis of the feedback they have got until then. Since clients exchange awareness information through the awareness server, only that server can provide the above feedback, i.e.,

$\forall\Box$((server = notifies-event) $\rightarrow$ $\forall\bigcirc$(server-provides-feedback-on-past-activities))

Feedback on past activities can be either an indication that some enacted events are going to be notified or an enumeration of those events. In the above design example, an indication might not make enough sense to designers. Enumeration instead might prompt them to suspend their actions until they get the notification and see what the events are about. Enumeration can be realized by counting how many events have been notified and how many are still along the way to the recipients. The server needs only to maintain a counter and increment it upon the release of each event, so as to inform clients of how many events they should still await. To avoid confusion with the counter variables of the encoding, we call this counter enum:

$\forall\Box$((server = notifies-event $\rightarrow$ $\forall\bigcirc$(server = increments-enum $\wedge$ server = notifies-enum)) (26)

$\forall\Box$((server = notifies-enum) $\rightarrow$ $\forall\Diamond$(client2 = receives-enum)) (27)

Yet we need also to decrement that counter upon the delivery of each event, otherwise the clients would await events which would have arrived already. Hence the above two formulae must be complemented by a third one, i.e.,

$\forall\Box$((client2 = receives-notification) $\rightarrow$ $\forall\bigcirc$(enum-is-decremented))

One way to realize the above would be by allowing the server to know whether notified events have actually reached a client so as to decrement the enum counter. This however requires synchronization among the clients and the server, which is impossible in the WWW. An alternative way would be to let the client decrement itself the counter upon the arrival of each event, i.e.,

$\forall\Box$((client2 = receives-notification) $\rightarrow$ $\forall\Diamond$(client2 = decrements-enum)) (28)

Below we examine the suitability of this alternative.

6.2.4. Model Checking Usability

We model checked whether event enumeration can improve the usability of awareness-support protocols, i.e., whether it can satisfy the fourth requirement of Section 3 (hence the word '*intended*' in that requirement was interpreted as '*a client should know exactly how many events s/he should await*'). Before doing this, we added the new values increments-enum and notifies-enum to the definition of the server variable, as well as the values receives-enum and decrements-enum to the definition of the client variable (the value decrements-enum was added to the definition of the server the first time). These values were incorporated then into two assignments of the server (in order to encode formulae (26) and (27)) and into another two assignments of the client (to encode formulae (27) and (28)). We also encoded the counter of events as follows:

```
VAR
    enum : 0 . . 50;
ASSIGN
    init(enum) := 0;
    next(enum) :=
    case
        (enum < 50) & (server = increments-enum):
        enum + 1;
        (enum = 50) & (server = increments-enum):
        50;
        (enum > 0) & (client2 = decrements-enum)
        enum – 1;
        1 : 0;
    esac;
```

Moreover, we allowed both clients to cause events and register interests in an interleaving fashion (as in Section 6.2.3), while we complicated further the dependencies between events with the following formula:

$\forall\Box$((client = receives-notification) $\rightarrow$ $\forall$[$\neg$(client = causes-event) U (enum = 0)]) (29)

This formula imposes a strict order on client activities in order to satisfy the aforementioned usability requirement. When we checked again the negation of formula (25) we received no indications of livelock, while the cases of delayed notification were also fewer. This implies that by incorporating the properties denoted by formulae (26)-(28) we can indeed improve usability, at least as concerns the scenarios represented in the encoding. We must point out nevertheless that the above solution should always be accompanied by timestamps, since the latter play a critical role in some cases (in cooperative design, for example, it would be useful to know the temporal order of notified events so as to appreciate the current state of design).

7. CONCLUSIONS AND FUTURE RESEARCH

Groupware systems can bring substantial benefits in the workplace, but when their development is driven by imprecise assumptions about time, they do not allow users to collaborate effectively. The aim of this paper is to enable groupware developers to revisit such assumptions in the early phases of development with the aid of model checking.

Specifically, this paper has examined the problem of group awareness within the context of the WWW. Because the WWW poses challenges that are not addressed adequately by current awareness protocols, we used formal methods to describe desired properties of these protocols and verify their correctness. Thanks to the SMV model checker we identified several scenarios of faulty behavior in two protocols that reflect the current state of practice in awareness support and we suggested subsequently a number of improvements. These improvements concern *event buffering*, *event timestamping* and *event enumeration*, and they emerged from specific counterexamples of the model checker. To the best of our knowledge, the above improvements are not incorporated currently in any protocol for awareness support.

Central to our work has been a methodology for encoding CTL formulae into the language of SMV. This methodology increases the efficiency of model checking and reduces the encoding effort sometimes, while it can be applied easily by developers since it saves them from the task of drawing state-transition models. We must note though that explicit model checkers, like EMC [17] and SPIN [33], perform better than SMV in the verification of software, because the space complexity of explicit model checking is proportional not to the number of possible states but to the number of reachable ones, which are usually fewer in software systems. EMC encodes however each state explicitly, thus making model checking laborious. SPIN, in turn, can only verify linear-time formulae, so it is unsuitable for the verification of groupware which allows several paths of synchronization at run time. Instead, the specification of groupware properties with CTL and their efficient verification thanks to the use of SMV and our encoding methodology justify our overall approach, and make evident that it fits better to groupware development than other approaches based on explicit model checkers.

Our own approach can also serve as a starting point for evaluating groupware usability with model checking. To this end, we aim to develop a generic framework for detecting usability constraints in CSCW. For example, because our approach can only verify properties known in advance but not properties that evolve dynamically, we aim to extend it towards *on-the-fly verification* [32] so as to enable the detection of usability constraints while collaboration is still in progress. Other approaches which also utilize model checking for evaluating usability (e.g., [36]) are very broad in scope and do not consider the semantics of cooperative work. Finally, we wish to establish a mathematical foundation for our encoding methodology, whereby we would be able to prove formally its equivalence with automata-based encodings.

REFERENCES

[1] G. Abowd and A. Dix. "Giving Undo Attention." *Inter. Comp.*, vol. 4, pp. 317-342, June 1992.

[2] T. Ahmed and A.R. Tripathi. "Static Verification of Security Requirements in Role-Based CSCW Systems", In *Proc. 8th SACMAT*, 2003, pp. 196-203. New York: ACM Press.

[3] D.I. Ballard and D.R. Seibold. "Time orientation and temporal variation across work groups: Implications for group and organizational communication." *Western J. Commun.*, vol. 64, pp. 218-242, 2000.

[4] S. Benford, C. Brown, G. Reynard and C. Greenhalgh. "Shared Spaces: Transportation, Artificiality and Spatiality," In *Proc. 6th CSCW*, 1996, pp. 77-86. New York: ACM Press.

[5] T. Berlage and A. Genau. "A Framework for Shared Applications with a Replicated Architecture," In *Proc. 6th UIST*, 1993, pp. 249-257. New York: ACM Press.

[6] T. Berners-Lee, R. Cailliau, A. Luotonen, H.F. Nielsen and A. Secret. "The World Wide Web." *Commun. ACM*, vol. 37, pp. 87-94, Aug 1994.

[7] R. Bharadwaj and C.L. Heitmeyer. "Model checking complete requirements specifications using Abstraction." *Automated Software Engineering*, vol. 6, pp. 37-68, Jan. 1999.

[8] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita and Y. Zhu. "Symbolic model checking using SAT procedures instead of BDDs," In *Proc. 36th DAC*, 1999, pp. 317-320. New York: ACM Press.

[9] R.E. Bryant. "Symbolic Boolean manipulation with ordered binary decision diagrams." *ACM Computing Surveys*, vol. 24, pp. 293-318, Sept 1992.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond." *Information Computation*, vol. 98, pp. 142-170, June 1992.

[11] Cadence. "SMV." Internet: www-cad.eecs.berkeley.edu/~kenmcmil/smv/ [May 22, 2007].

[12] J.J. Cadiz, G. Venolla, G. Jancke and A. Gupta. "Designing and Deploying an Information Awareness Interface," In *Proc. 9th CSCW*, 2002, pp. 314-323. New York: ACM Press.

[13] M. Cataldo, P.A. Wagstrom, J.D. Herbsleb and K.M. Carley. "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools," In *Proc. 11th CSCW*, 2006, pp. 353-362. New York: ACM Press.

[14] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin and J.D. Reese. "Model Checking Large Software Specifications." *IEEE Trans. Software Eng.*, vol. 24, pp. 498-520, July 1998.

[15] M. Chechik, B. Devereux, S. Easterbrook and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking." *ACM Transactions on Software Engineering and Methodology*, vol. 12, pp. 371-408, Oct 2003.

[16] E.M. Clarke and E.A. Emerson. "Design and synthesis of synchronization skeletons using branching-time temporal logic," In *Proc. IBM Workshop on Logic of Programs*, 1981, LNCS 131, pp. 52-71. Berlin: Springer.

[17] E.M. Clarke, E.A. Emerson and A.P. Sistla. "Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications." *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 244-263, April 1986.

[18] E.M. Clarke, O. Grumberg and D.A. Peled. *Model Checking*. Cambridge, MA: MIT Press, 1999.

[19] E. Clarke, D. Kroening, J. Ouaknine and O. Strichman. "Computational challenges in bounded model checking." *Int. J. Software Tools for Technol. Transfer*, vol. 7, pp. 174-183, April 2005.

[20] A. Dix. "LADA – A logic for the analysis of distributed actions," In *Proc. 1st Eurographics*, 1994, pp. 317-332. New York: Springer.

[21] P. Dourish and V. Bellotti. "Awareness and Coordination in Shared Workspaces," In *Proc. 4th CSCW*, 1992, pp. 107-114. New York: ACM Press.

[22] D. Drusinsky and D. Harel. "On the Power of Bounded Concurrency I - Finite Automata." *J. ACM*, vol. 41, pp. 517-539, May 1994.

[23] C.A. Ellis, S.J. Gibbs and G.L. Rein. "Groupware: Some Issues and Experiences." *Commun. ACM*, vol. 34, pp. 39-58, Jan 1991.

[24] C.A. Ellis. "Team Automata for Groupware Systems," In *Proc. GROUP '97*, 1997, pp. 415-424. New York: ACM Press.

[25] N. Francez. *Fairness*. New York: Springer, 1986.

[26] P. Godefroid, J.D. Herbsleb, L.J. Jagadeesan and D. Li. "Ensuring Privacy in Presence Awareness Systems: An Automated Verification Approach," In *Proc. 8th CSCW*, 2000, pp. 59-68. New York: ACM Press.

[27] T.C.N. Graham. "GroupScape: Integrating Synchronous Groupware and the World Wide Web," In *Proc. INTERACT '97*, 1997, pp. 547-554. London: Chapman and Hall.

[28] S. Gundavaram. *CGI Programming on the World Wide Web*. Sebastopol, CA: O'Reilly, 1996.

[29] C. Gutwin. "The Effects of Network Delays on Group Work in Real-Time Groupware," In *Proc. 7th ECSCW*, 2001, pp. 299-318. Dordrecht, The Netherlands: Kluwer.

[30] C. Gutwin and S. Greenberg. "Effects of Awareness Support on Groupware Usability," In *Proc. CHI '98*, 1998, pp. 511-518. New York: ACM Press.

[31] J. Hill and C. Gutwin. "Awareness support in a groupware design toolkit," In *Proc. GROUP '03*, 2003, pp. 258-267. New York: ACM Press.

[32] G. Holzmann. "On-the-fly model checking." *ACM Computing Surveys*, vol. 28, Dec 1996.

[33] G.J. Holzmann. "The Model Checker SPIN." *IEEE Trans. Software Eng.*, vol. 23, pp. 279-295, May 1997.

[34] A. Imine, P. Molli, G. Oster and M. Rusinowitch. "Proving correctness of transformation functions in real-time groupware," In *Proc. 8th ECSCW*, 2003, pp. 277-294. Dordrecht, The Netherlands: Kluwer.

[35] Imine, M. Rusinowitch, G. Oster and P. Molli. "Formal design and verification of operational transformation algorithms for copies convergence", *Theoretical Computer Science*, vol. 351, *Special Issue on Algebraic Methodology and Software Technology*, pp. 167-183, Feb 2006.

[36] CCTC. "IVY: a model-based usability analysis environment." Internet: http://www.di.uminho.pt/projects/ivy [July 9, 2007].

[37] Johnson. "A formal approach to the representation of CSCW systems," In *Proc. HCI '93*, 1993, pp. 335-352. Cambridge, UK: Cambridge University Press.

[38] O. Kupferman, M.Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Branching-Time Model Checking." *J. ACM*, vol. 47, pp. 312-360, March 2000.

[39] Lushman and G.V. Cormack. "Proof of correctness of Ressel's adOPTed algorithm", *Inform. Proc. Lett.*, vol. 86, pp. 303-310, June 2003.

[40] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems - Safety*. Berlin: Springer, 1995.

[41] J.E. McGrath. "Time matters in groups," In *Intellectual Teamwork: Social and Technical Foundations of Cooperative Work* (J. Galegher, R.E. Kraut and C. Egido, Eds.), pp. 23-61. London: Lawrence Erlbaum Associates, 1990.

[42] K.L. McMillan. *Symbolic Model Checking*. Norwell, MA: Kluwer, 1993.

[43] P. Palanque and R. Bastide. "Formal specification and verification of CSCW using the Interactive Cooperative Object formalism," In *Proc. HCI '95*, 1995, pp. 213-231. Cambridge, UK: Cambridge University Press.

[44] K. Palfreyman and T. Rodden. "A protocol for user awareness on the World Wide Web," In *Proc. 6th CSCW*, 1996, pp. 130-139. New York: ACM Press.

[45] Papadopoulos. "An Extended Temporal Logic for CSCW." *Comp. J.*, vol. 45, pp. 453-472, Aug. 2002.

[46] C. Papadopoulos. "An Automata-based Approach to CSCW Verification." *Int. J. Cooperative Information Systems*, vol. 13, pp. 183-209, June 2004.

[47] M.R. Prasad, A. Biere and A. Gupta. "A survey of recent advances in SAT-based formal verification." *Int. J. Software Tools Technol. Transfer*, vol. 7, pp. 174-183, April 2005.

[48] V. Schuppan and A. Biere. "Verifying the IEEE 1394 FireWire Tree Identify Protocol with SMV." *Formal Aspects of Computing*, vol. 14, pp. 267-280, April 2003.

[49] Stotts and R. Furuta. "Interpreted collaboration protocols and their use in groupware prototyping," In *Proc. 5th CSCW*, 1994, pp. 121-131. New York: ACM Press.

[50] D. Stotts and J. Navon. "Model Checking CobWeb Protocols for Verification of HTML Frames Behavior," In *Proc. WWW '02*, 2002, pp. 182-190. New York: ACM Press.

[51] L. Suchman. *Plans and Situated Actions*. Cambridge, UK: Cambridge University Press, 1987.

[52] C. Sun, X. Zia, Y. Zhang, Y. Yang and D. Chen. "Achieving Convergence, Causality-Preservation and In-tention-Preservation in Real-Time Cooperative Editing Systems." *ACM Transactions on Comp.-Hum. Interact.*, vol. 5, pp. 63-108, March 1998.

[53] A.S. Tanenbaum. *Computer Networks (4th Ed.)* Upper Saddle River, NJ: Prentice-Hall, 2003.

[54] M.H. ter Beek, C.A. Ellis, J. Kleijn and G. Rozenberg. "Synchronizations in Team Automata for Groupware Systems." *Comp. Support. Coop. Work*, vol. 12, pp. 21-69, Feb 2003.

[55] M.H. ter Beek, M. Massink, D. Latella and S. Gnesi. "Model Checking Groupware Protocols," In *Cooperative Systems Design – Scenario-based Design of Collaborative Systems* (F. Darses, R. Dieng, C. Simone and M. Zacklad, Eds.), vol. 107 of *Frontiers of Artificial Intelligence and Applications*, pp. 179-194. Amsterdam: IOS Press.

[56] M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri and M. Sebastianis. "A Case Study on the Automated Verification of Groupware Protocols," In *Proc. ICSE'05*, 2005, pp. 596-603. New York: ACM Press.

[57] M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri and M. Sebastianis. "Model Checking Publish/Subscribe Notification for thinkteam," In *Proc. FMICS'04*, 2005, pp. 275-294. ENTCS 133.

[58] M.H. ter Beek, M. Massink and D. Latella. "Towards Model Checking Stochastic Aspects of the thinkteam User Interface," In *Proc. DSVIS'05*, 2006, LNCS 3941, pp. 39-50, Berlin: Springer.

[59] M.H. Tran, Y. Yang and G.K. Raikundalia. "F@: A Framework of Group Awareness in Synchronous Distributed Groupware," In *Proc. APWeb*, 2006, LNCS 3841, pp. 461-473, Heidelberg, Germany: Springer.

[60] J. Trevor, T. Koch and W. Wötzel. "MetaWeb: Bringing Synchronous Groupware to the World Wide Web," In *Proc. 5th ECSCW*, 1997, pp. 65-80. Dordrecht, The Netherlands: Kluwer.

[61] P. Wolper. "Constructing Automata from Temporal Logic Formulae: A Tutorial," In *Lectures on Formal Methods and Performance Analysis* (E. Brinksma, H. Hermanns and J.-P. Katoen, Eds.), 2001, pp. 261-277. LNCS 2090, Berlin: Springer.

[62] B. Yang. *SMV 2.4b*. Internet: www.cs.cmu.edu/~bwolen/software/smv/ [Nov. 23, 2006].

Received: June 15, 2007 Revised: July 09, 2007 Accepted: July 10, 2007

APPENDIX

A.1. Computation Tree Logic

Temporal operators appearing in CTL formulae must be preceded either by the *universal* ('∀') or the *existential* ('∃') quantifier, which denote respectively that a property holds in every path out of a state or along some specific path(s) only. Moreover, past-time operators are not allowed, so only the operators '○', '◊', '□', 'U' and 'R' may appear in CTL formulae. The *next* operator '○' is used to denote that a property holds in the immediately following state, the *eventually* operator '◊' that a property will hold at some future state, and the *henceforth* operator '□' that a property will hold at every state hereafter. The *until* operator 'U', in turn, is used between two propositions to denote that there will be a state where the second proposition will start holding while the first one will have held at every state prior to that state. The *releases* operator 'R', finally, is the dual of 'U' and is used between two propositions to denote that there will be a state where the first proposition will start holding and the second one will become False.

The possible combinations of temporal operators with path quantifiers can be defined in CTL as follows: $f \vee g \equiv \neg (\neg f \wedge \neg g)$, $\forall \Diamond g \equiv \forall[\text{True U } g]$, $\exists \Diamond g \equiv \exists[\text{True U } g]$, $\forall \Box f \equiv \neg \exists[\text{True U } \neg f]$,

$\exists[f \text{ U } g] \equiv \exists[g \text{ R } f]$ and $\exists \Box f \equiv \neg \forall[\text{True U } \neg f]$.

Last, the syntax of CTL formulae can be defined by the following rules:

[i] Every atomic proposition is a CTL formula.

[ii] If f and g are CTL formulae, then so are $\neg f$, $f \wedge g$, $\forall \bigcirc f$, $\exists[f \text{ U } g]$ and $\forall[f \text{ U } g]$.

A.2. Kripke structures

A Kripke structure is a quadruple (S, S_0, T, L) over a set of atomic propositions (i.e., Boolean variables). S is a finite set of states, which are semantic concepts that do not appear explicitly in formulae. Each state is an assignment of values to system variables. S_0 in turn, is a subset of S containing initial states, while T is a transition relation between states (i.e., $T \subseteq S \times S$) such that each state in S is accessible by at least one other state. L, finally, is a function that labels each state in S with the set of propositions which are True in that state.

A.3. Symbolic Algorithms

Each state of a Kripke structure is encoded in symbolic algorithms by an assignment of Boolean values to the set of state variables, while transitions are expressed as Boolean formulae that manipulate variables from the originating and the final state. For example, given the predicate $p_0(s) \equiv (\exists s : s \in S_0)$, where S_0 denotes the set of initial states in a Kripke structure, and the predicate $p_R(s, t) \equiv (\exists s : s \in S \wedge t \in S \wedge (s, t) \in T \wedge \neg(\exists r : r \in S : (s, r) \in T \wedge (r, t) \in T)$, where S and T denote respectively the set of states and the transition relation of the Kripke structure (so p_R is True if t is accessible by s with only one transition), then the set of states in the Kripke structure that are accessible by the initial states with one transition can be expressed by the predicate $p_1(s) \equiv (\exists x : x \in S : p_0(x) \wedge p_R(x, s))$. Similarly, the possibility of error in the behaviour of a system can be detected by computing iteratively the predicates $p_1(s), \ldots, p_k(s)$, where $p_k(s) \equiv (\exists x : x \in S : p_{k-1}(x) \wedge p_R(x, s))$, and intersecting at each step the sets corresponding to these predicates by a set of undesirable states (i.e., states indicating faulty behaviour).

A.4. SMV Basics

As we mentioned in the main text, SMV was the first model checker that incorporated symbolic algorithms. This checker represents states and transitions *via* BDDs. Today there exist quite a few versions of SMV (e.g., [11], [62]), but the features presented in this section are common to all versions.

A finite-state system can be described in SMV with one or more *modules*, which represent concurrently executing processes and are defined by the keyword MODULE followed by a string. These processes can be instantiated several times, take on parameters, and reference variables declared in other modules (like the procedures and functions of structured programming languages). The concurrent execution of SMV modules can be done either synchronously or in an interleaving fashion. In the latter case, the keyword process must precede the declaration of module instances. Modules contain also variable declarations denoted by the keyword VAR, which may refer to Boolean variables, enumerated symbolic types, variables within an integer subrange, or arrays. Variables may also take values from sets or expressions involving logical connectives. For example, taking advantage of SMV's macro-definition facility (i.e., DEFINE declarations), one can define new symbols from Boolean expressions but may not define new system variables.

Modules are used to specify transition relations, which are written inside ASSIGN statements in the form of Boolean expressions and/or parallel assignments. In both cases, the variables of each module are evaluated in a next state according to their values in the current state. For example, the statement ASSIGN next(x) := !x sets the next-state value of the Boolean variable x to the negation of its current value. Next-state values may be defined also through case statements nested within an ASSIGN statement. The branches of case statements are examined in the order of their appearance and the first one that evaluates to True is chosen for execution. The last branch ensures that no transition is made if none of the previous branches is True. Before the definition of a next-state value, ASSIGN statements define the corresponding initial-state value. So, for any variable y, init(y) refers to the value of y in the initial state of the Kripke structure. A significant advantage of SMV regarding the specification of reactive systems is its ability to represent non-deterministic transitions. For example, when an ASSIGN

statement associates a variable with a set, the value of that variable is set randomly to a member of that set (e.g., ASSIGN next(x) := {0, 1}). Another way to specify transition relations is through TRANS declarations, which include expressions defining these relations as propositions. The essential difference between TRANS and ASSIGN statements is that the former define transition relations declaratively, whereas the latter define them imperatively.

Last, the formulae to be verified are written as SPEC statements, while fairness constraints are written as CTL formulae that are preceded by the keyword FAIRNESS.

A.5. Encoding of Palfreyman and Rodden's Protocol

Based on the above, we wrote two scenarios for Palfreyman and Rodden's protocol (as we have already mentioned), i.e., one involving five clients and another ten. Below we present an example scenario with only one client:

```
MODULE main
VAR
    client: {idle, sends-query, not-send-query, gets-results, signs-on, not-sign-on, signs-off, in-tr1, in-tr2, in-tr3, in-tr4, in-tr5};
    server: {idle, gets-query, sends-results, gets-sign-on, gets-sign-off, in-tr6, in-tr7, in-tr8, in-tr9, in-tr10, in-tr11};
    cntr1: 1 . . 1000;
    ;; the same for all the other counter variables
ASSIGN
    init(client) := idle;
    init(server) := idle;
    init(cntr1) := 1;
    ;; the same for all the other counter variables
next(client) :=
  case
    (client = gets-results) : {idle, gets-results, signs-on, signs-off, sends-query};
    (client = signs-on) & (cntr1 = 1) : {idle, sends-query, in-tr1};
    (client = idle) & (cntr2 = 1) : {idle, in-tr2};
    (client = in-tr2) & (cntr2 < 1000) : {sends-query, signs-on, signs-off, gets-results, in-tr2};
    (client = in-tr2) & (cntr2 = 1000) : {sends-query, signs-on, signs-off, gets-results};
    (client = signs-off) & (cntr3 = 1) : in-tr3;
    (client = in-tr3) & (cntr3 < 1000) : {in-tr3, signs-on};
    (client = in-tr3) & (cntr3 = 1000) : signs-on;
    (client = sends-query) & (cntr4 = 1) : in-tr4;
    (client = in-tr4) & (cntr4 < 1000) & (cntr5 = 1) : {in-tr4, not-send-query};
    (client = in-tr4) & (cntr4 = 1000) & (cntr5 = 1) : not-send-query;
    (client = not-send-query) & (cntr5 < 1000) : {not-send-query, gets-results};
    (client = not-send-query) & (cntr5 = 1000) : gets-results;
    (client = in-tr1) & (cntr1 < 1000) & (cntr6 = 1) : {in-tr1, not-sign-on};
    (client = in-tr1) & (cntr1 = 1000) & (cntr6 = 1) : not-sign-on;
    (client = not-sign-on) & (cntr6 < 1000) : {not-sign-on, signs-off};
    (client = not-sign-on) & (cntr6 = 1000) : signs-off;
    (server = sends-query) & (cntr7 = 1) : in-tr5;
    (server = in-tr5) & (cntr7 < 1000) : {in-tr5, gets-results};
    (server = in-tr5) & (cntr7 = 1000): gets-results;
    1 : idle;
```

```
    esac;
next(server) :=
    case
        (server = gets-query) : sends-results;
        (server = sends-results) : {gets-query, idle, gets-sign-on, gets-sign-off};
        (server = gets-sign-on) & (cntr8 = 1) : {idle, in-tr6};
        (server = in-tr6) & (cntr8 < 1) : {in-tr6, gets-query, gets-sign-on, gets-sign-off};
        (server = in-tr6) & (cntr8 = 1000) : {gets-query, gets-sign-on, gets-sign-off};
        (server = gets-sign-off) & (cntr9 = 1) : {idle, in-tr7};
        (server = gets-sign-off) & (cntr9 < 1000) : {in-tr7, gets-sign-on, gets-sign-off};
        (server = gets-sign-off) & (cntr9 = 1000) : {gets-sign-on, gets-sign-off};
        (client = signs-on) & (cntr10 = 1) : in-tr8;
        (server = in-tr8) & (cntr10 < 1000) : {in-tr8, gets-sign-on};
        (server = in-tr8) & (cntr10 = 1000) : gets-sign-on;
        (client = signs-off) & (cntr11 = 1) : in-tr9;
        (server = in-tr9) & (cntr11 < 1000) : {in-tr9, gets-sign-off};
        (server = in-tr9) & (cntr11 = 1000) : gets-sign-off;
        (client = sends-query) & (cntr12 = 1) : in-tr10;
        (server = in-tr10) & (cntr12 < 1000) : {in-tr10, gets-query};
        (server = in-tr10) & (cntr12 = 1000) : gets-query;
        (server = idle) & (cntr13 = 1) : in-tr11;
        (server = in-tr11) & (cntr13 < 1000) : {in-tr11, idle, gets-query, gets-sign-on, gets-sign-off};
        (server = in-tr11) & (cntr13 = 1000) : {idle, gets-query, gets-sign-on, gets-sign-off};
        1 : idle;
    esac;
next(cntr1) :=
    case
        (client = in-tr1) : cntr1 + 1;
         1 : cntr1;
    esac;
;; the same for all the other counter variables except cntr5 and cntr6
next(cntr5) :=
    case
        (cntr4 = 1000) & (client = not-send-query) : cntr5 + 1;
         1 : cntr5;
    esac;
next(cntr6) :=
    case
        (cntr1 = 1000) & (client = not-sign-on) : cntr6 + 1;
         1 : cntr6;
    esac;
. . .
SPEC AG((client = signs-on) –> E[(client = sends-query) U (client = signs-off)])
```