Specifying Internet applications with *DiCons*

J.C.M. Baeten
Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
josb@win.tue.nl

H.M.A. van Beek
Eindhoven Embedded Systems Institute (EESI), Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
harm@win.tue.nl

S. Mauw
Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
sjouke@win.tue.nl

Keywords
Internet applications, language design, distributed consensus, *DiCons*.

ABSTRACT
It is not easy to build Internet applications with common techniques, such as CGI scripts and Perl. Therefore, we designed the *DiCons* language, which supports the development of a range of Internet applications at the appropriate level of abstraction. In this paper we discuss the design of *DiCons*, we give an overview of the tool support and we explain the language by means of an example.

1. INTRODUCTION
Some trends concerning the development of new Internet applications can be observed. First of all, the Internet and applications of Internet are developed with a tremendous speed. The first to come with an interesting application sets the standard for that application area. Many new services are realized, for example applications which support auctions or voting via Internet.

Secondly, large portals replaced the *old style* search engines. They provide functionality that goes beyond mere guidance through the Internet. The longer that the visitor stays at the portal site and the more often that he uses functionality provided by the portal, the higher the income from advertisements will be. Therefore, portals must offer interesting applications and must keep their functionality up to date. This does not only imply that portals must maintain a large set of applications, but also that they must be able to rapidly develop new services. Short *time to market* is an important asset.

The third observation is that the number of commercial transactions on the Internet is growing. Security and dependability are important factors at all levels of interaction. Apart from proper use of cryptographic techniques, this also requires that the protocols by which information is exchanged are correct. A voting system, e.g., must guarantee that the winner is actually the candidate that has received most support.

We want to be able to quickly develop secure and dependable Internet applications. Some problems that occur are, firstly, that several languages are involved, such as html, cgi scripts, and other scripting languages. Secondly, the level of abstraction of the language used often does not correspond to the level on which we think about an application: there is no C-primitive for filling out a Web form. Thirdly, current practices do not lend themselves to validation or verification.

Thus, our goal is to develop a language at the right level of abstraction, that is amenable to (formal) validation or verification. In order to make the problem more concrete and the solution more feasible, we limit the class of applications we consider. First of all, we consider applications where several users strive to reach a common goal without having to meet. We call this kind of applications *distributed consensus applications*. A central location on an Internet server should support this. We are interested in asynchronous communication, as exemplified by the sending of e-mails and Web forms. Users do not communicate directly but only communicate with the central application. Finally, we only want to use standard techniques, so the user does not require special programs, software or plug-ins. An Internet connection, e-mail and a Web browser should suffice, on any hardware/software platform.

In this paper a new specification language *DiCons* (*Distributed Consensus*) is introduced to specify Internet applications for *distributed consensus*. Major characteristic of this class of protocols is that a number of users strive to reach a common goal (e.g. make an appointment, evaluate a paper, select a "winner"). The problem is that the users do not want to physically meet to solve their goal, nor will there be any synchronized communications between the users. A central system, viz. an Internet application, must be used to collect and distribute all relevant information.

This class of applications was the starting point for developing our language. The language must both be expressive

enough and concrete. In order to be applicable to an appropriate range of problems, it must have the right expressive power. The language must be concrete enough, such that automatic generation of an executable is feasible.

Typical examples of applications that our research targets at, are: Meeting scheduler, election support system, auction, and gift selection. These examples have in common that they support a task which is algorithmically simple but requires many interactions. This task is taken over by a central application, handling all interactions with the users. In this paper, we illustrate this by the example of a gift selection system.

The purpose of this paper is to give a description of the language *DiCons* and the tools developed for it. To this end, we first give an overview of the design decisions we took in order to arrive at this language and its prototype tools. Next, we illustrate the use of *DiCons* by the example of the gift selection system. After that, we compare our approach with other methods and techniques. Finally, we finish with some concluding remarks and ideas for the further development of the language and tools.

2. DESIGN OF *DICONS*

In this section we will discuss the considerations that led to the current design of the *DiCons* language and we describe the basic ingredients of *DiCons*.

2.1 Restrictions

In order to not have to face the complete problem of writing Internet applications in general we restrict our problem setting in several ways. First of all, we focus on a class of applications which is amenable to formal verification with respect to behavioral properties. This means that the complexity of the application comes from the various interactions between users and a system, rather than from the data being exchanged and transformed. Implications for the design of the language are that the primitive constructs are *interactions*, which can be composed into complex behavioral expressions. Furthermore, it implies that the development of the language and its formal semantics must go hand in hand. Nevertheless, we will not discuss semantical issues in the current paper.

A further restriction follows from the assumption that although the users work together to achieve some common goal, there will be no means for the users to communicate directly with each other. We assume a single, central application that follows a strictly defined protocol in communication with the users.

The last consideration with respect to the design of *DiCons* is that we want to make use of standard Internet technology only. Therefore, we focus on communication primitives such as e-mail and Web forms. This means that a user can interact with the system with a standard Web browser, without the need for additional software such as plug-ins. Of course, it must be kept in mind that the constructs must be so general as to easily support more recent developments, such as ICQ or SMS messages. Currently, we only consider asynchronous communication between client and server.

2.2 Overview of language constructs

Bearing above considerations with respect to the application domain and available technology in mind, we come to a description of the basic constructs of *DiCons*. We will first list the language ingredients and later discuss these in more detail, without precisely defining their syntax and semantics. The example in Section 4 will serve to show the flavor of the *DiCons* syntax and the way in which the language can be used.

users and roles The first observation is that, since an application may involve different users, the application must be able to identify users. Moreover, since different users may want to use the system in the same way, it must be possible to group users into so-called *roles*.

interactions We have to identify the communication primitives, which we will call *interactions*. They form the basic building blocks of the behavioral descriptions. Interactions are abstract descriptions which are identified by their name and may carry input and output parameters.

behavior A number of interactions with the same user may be combined to form a *session*. Sessions and interactions can be composed into complex behavioral descriptions which define an *application*.

presentations The abstract interactions are represented to the user by means of concrete communication means, such as e-mail and Web forms. This is called the *presentation* of an interaction.

data In order to transform (user) data and keep state information, we need a means to define and manipulate data (expressions, variables, data structures, etc.)

2.3 Users and roles

A user is an entity that can interact with the system. A user has three attributes: a name (for reference), an e-mail address (in case e-mail communication is desired), and a password (in case user authentication is needed). Users are grouped according to their role. Users with the same role are offered the same interaction behavior. In *DiCons* roles can be defined and variables can be declared which denote users with a given role.

2.4 Interactions

The basic problem when defining the interaction primitives is to determine the right level of abstraction. Taking, e.g., an http request as a primitive interaction will lead to programs which are too detailed. On the other hand, if we would define a complete user session as a primitive interaction, we could not deal with the variety of different sessions that occur in an application.

In order to get a feeling of the level of abstraction which is optimally suitable, look at Figure 1. In this drawing we sketch a typical scenario of an Internet application which is called the *Meeting Scheduler* (see [22]). This is an application which assists in scheduling a meeting by keeping track of all suitable dates and sending appropriate requests and convocations to the intended participants of the meeting.

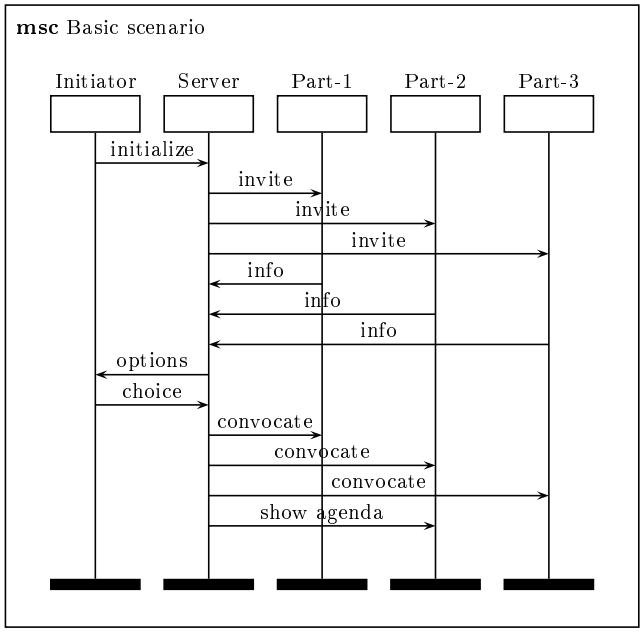

Figure 1: A scenario of an Internet application

The drawing is a so-called *Message Sequence Chart* (MSC, see [17]), which is a standardized visual language, especially suited for requirements engineering. The example shows that we have two roles, viz. initiator and participant. In this scenario, there is only one user with role initiator, while there are three users with role participant. The MSC shows that the initiator starts the system by providing it with meeting information. Next, the system sends an invitation to the participants who reply by stating which dates suit them. After collecting this information, the system informs the initiator about the options for scheduling the meeting and awaits the choice made by the initiator. Finally, the system informs the participants about the date and offers the users to have a look at the agenda. Only participant 2 is interested in the agenda.

This example nicely shows at which level of detail one wants to specify such an application. The arrows in the diagram represent the basic interaction primitives. First, look at the *invite* messages. Since the participants don't know that they will be invited for a meeting, the initiative of this interaction is at the server side. The way in which a server can actively inform a client is by sending an e-mail. This interaction only contains information transmitted from the server to the user. The messages *options* and *convocate* are also implemented as e-mails.

Next, look at message *info*. This interaction is initiated by the user and is best implemented as a Web form supplied by the server, on request of the user and filled in by the user. The message *choice* also stands for a Web form being filled in.

The last message, *show agenda* contains information sent by the server to the user, on request of the user. This is simply the request and transmission of a non-interactive Web page.

Finally, we look at the first message, *initialize*. The initiator has to supply the system with various kinds of information, such as a list of proposed dates and a list of proposed participants. This will probably be implemented as a dialogue between the user and the system in the form of a series of Web forms. This is called a *session*.

We summarize the three basic interaction schemes in Figure 2. Notice that the third scheme, the session, consists of a series of more primitive interactions. It starts with a client requesting a form and submitting it after having it filled in. This is the interaction which starts the session. Next, comes a series of zero or more submissions of Web forms. These are interactions which come in the middle of a session. And, finally, the session ends with the server sending a simple Web page after the last submission of the client.

In *DiCons* we have constructs for these five interaction primitives. We have used a naming scheme for the interaction primitives which is based on their properties. First, we make a distinction based on the flow of information. If the information goes from the server to the client, we call this a *server push*, while if the information flows to the server, we call this a *server pull*. Notice that we reason from the viewpoint of the server in this respect.

The second distinction which we make is on which party takes the initiative for the interaction. Still reasoning from the viewpoint of the server we consider an *active* communication, which means that the server takes the initiative, a *reactive* communication, which means that the client takes the initiative, and a *session oriented* communication, which means that the communication is a response from the server to a prior submission of a Web form by the client.

Finally, notice that we extend the interaction primitives with parameters to express which information is being transmitted. An output parameter denotes information sent by the server to the client, while an input parameter is a variable in the data space of the server which will contain the information sent by the client to the server.

The notation for our communication primitives is given below.

active server push The server takes the initiative to send information (for o_i $(0 \leq i \leq n)$ output parameters):

mail to *client* $\leftarrow$ message$(o_0, \ldots, o_n)$

reactive server push The server sends a Web page on request of the client (for o_i $(0 \leq i \leq n)$ output parameters):

client $\leftarrow$ message(**out** $o_0, \ldots,$ **out** o_n)

reactive server pull The server sends a Web form on request of the client. After that, the client submits the filled in form. This interaction denotes the starting of a session. (for i_k $(0 \leq k \leq m)$ input parameters, o_k $(0 \leq k \leq n)$ output parameters and v_k $(0 \leq k \leq p)$ input/output parameters):

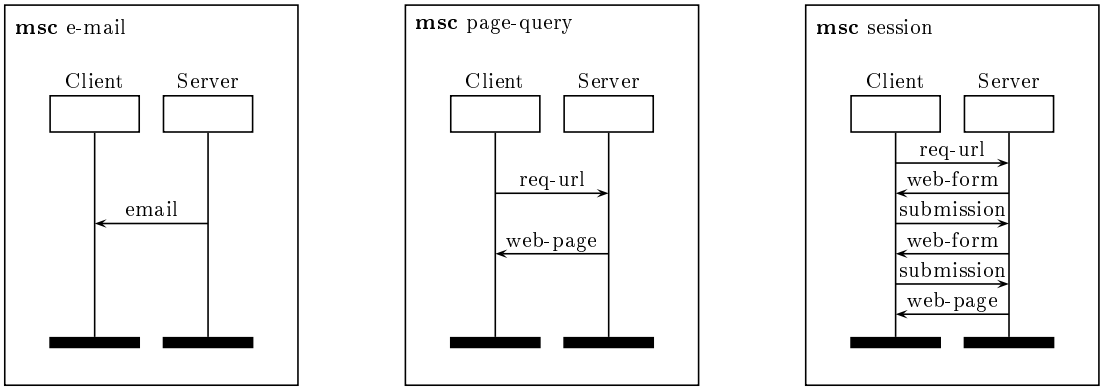

Figure 2: Interaction primitives

start session of *client* → message(**in** $i_0, \ldots$, **in** i_m, **out** $o_0, \ldots$, **out** o_n, **var** $v_0, \ldots$, **var** v_p)

session-oriented server pull The server sends a Web form to the client as a response to a prior form submission by the client. After that, the client submits the filled in form. This interaction is repeated in the middle of a session. (for i_k $(0 \leq k \leq m)$ input parameters, o_k $(0 \leq k \leq n)$ output parameters and v_k $(0 \leq k \leq p)$ input/output parameters):

session of *client* → message(**in** $i_0, \ldots$, **in** i_m, **out** $o_0, \ldots$, **out** o_n, **var** $v_0, \ldots$, **var** v_p)

session-oriented server push The server sends a non-interactive Web page to the client in response to a prior form submission by the client. This interaction is the last interaction of a session. (for o_i $(0 \leq i \leq n)$ output parameters):

end session of *client* ← message(**out** $o_0, \ldots$, **out** o_n)

Please notice that in our list of interaction primitives we did not mention the *active server pull*. The reason for this is simply that with standard Internet technology this interaction cannot be implemented. A Web server cannot take the initiative to obtain information from a client.

2.5 Behavior

Now that we have defined the basic interaction primitives, we can discuss the means to compose them into sessions and applications. An application describes the protocol to be executed by the server. A number of standard programming language constructs are supported in *DiCons*. We mention the following: sequential composition (denoted by a semi-colon), conditional branching (if-then-else-fi), repetition (for-all-do-od, and while-do-od, which is a parallel repetition), Since in most applications that we have studied users have to react before a given deadline, we have included a time-out construct in *DiCons* (until-do-od, which means that the body of this expression may execute until the given deadline). Finally, in order to manipulate the internal state of the application, we have assignments to variables and procedure calls. A session is simply a program fragment with the requirement that execution starts with a session-start interaction and ends with a corresponding session-end interaction.

2.6 Presentations

The interactions which are composed into a *DiCons* application are abstract in the sense that they only carry a name and possibly some parameters. Additional information is needed to determine how the interaction is implemented. In case of an e-mail, we need to specify the addresses of the sender and the receiver, the subject field, the body text and the places where the values of the output parameters must be filled in.

In case of a Web form, we must also define the fields where the user can type in values which are stored in the input parameters of the interaction. Furthermore, *DiCons* supports the inclusion of Java scripts which can put syntactic restrictions on the input provided by the user. Other supported features are pull-down selection menus, submit buttons, radio buttons and check boxes.

2.7 Data

Storing and manipulating data occurs at several places in a *DiCons* application. Therefore, a well equipped data language must be part of *DiCons*. Many programming languages have been developed to support the manipulation of data, so, rather than developing our own dedicated data language, we decided to include an existing language, namely Java [12]. The main reason for selecting Java, lies in its popularity in the Internet community, but also implementation issues made us decide for Java (because we use Java servlets, see Section 3).

In order to make *DiCons* as independent from the chosen data language as possible, we have defined the language in such a way that the included data language and the other parts of the language are orthogonal. Java fragments are only allowed in the definition of functions and procedures. Interaction with the other parts of the language takes place by calling these functions. In this way, Java can be easily substituted by other languages, such as C.

3. TOOLS FOR *DICONS*

We make use of several existing (Internet) techniques. First of all, we make use of Java servlets [16, 28]. These servlets generate HTML pages and HTML forms [25]. If data constraints are included into Web form, these constraints are checked by a piece of JavaScript code [14] which makes use of the regular expression, specified in the Perl/JavaScript regular expression syntax [11].

3.1 JavaCC

To implement a parser we have chosen to use the Java parser generator *Java Compiler Compiler (JavaCC)* [23]. This choice is made because we are specifying an Internet application and Java is the Internet specification language par excellence. JavaCC is a parser generator that produces parsers in Java from grammar specifications written in a lex/yacc-like manner.

We have implemented a package of classes which specify the different parts of the language: roles, types, variables, functions, interactions, sessions and the execution. After parsing an application, an object of type *DiConsApplication* is created. This object consists of different objects, all specifying one part of the application. These objects all have a method to convert that specific part of the application to a piece of Java code. By putting all these pieces together we get a Java application, viz. a Java servlet. This servlet can be compiled to Java byte-code by using a regular Java compiler. The file containing Java byte-code can be interpreted by a Web server.

3.2 Technical Aspects

In this section we will discuss some aspects of our specification which are non-trivial to implement. Since we cannot have multiple executions of one single servlet simultaneously we have to implement some kind of instance management and session management. The problem is that we want to be able to start several instances of some *DiCons* application. These instances must run independently and have disjoint state spaces. Within an instance of an application, several users may start parallel or overlapping sessions. Such sessions share the same data space.

3.2.1 Instance management

The servlet API does not implement instance management in the way we need it. It implements sessions using client-side cookies. Since one instance might concern more than one client, cookies cannot solve our instance management.

We introduce a new class *ServletInstance* in which all data concerning one instance can be stored. Furthermore, we add a variable containing the collection of available instances to our servlet. Each instance gets its own unique identifier. If one accesses the servlet without referring to an instance, a new instance is created. During a session, all Web forms are extended with a hidden variable containing the corresponding instance identifier. Submitting a Web form now results in including this identifier in the posted data. One can also call a servlet using a rewritten URL. This URL is extended with a query string containing an instance identifier. Calling the servlet like this results in continuing an instance if this is possible. If this instance does not exist the instance identifier is ignored and a new instance with a new, unique identifier is create. This identifier is composed of a letter I followed by eight randomly chosen digits.

Each time a servlet is called it checks whether an instance identifier is passed. If so, it tries to load that instance's data and continue the instance's execution. Otherwise, a new instance is created.

3.2.2 Session management

Session management support is built into the servlet API by using cookies. However, these techniques do not answer our needs. By using cookies we do not have the ability to run multiple simultaneous sessions using one and the same Web browser. Though this is not such a big shortcoming, one can turn off cookie usage in most of the Web browsers. This cookie problem can quite easily be solved by implementing sessions in the same way as we implemented instances.

We introduce a new class *ServletSession*. Sessions specified in the session part of the application are implemented as subclasses of this class. Sessions all have a session identifier which is unique for the instance it takes part in. This identifier is composed of a letter S followed by eight randomly chosen digits.

Again, we extend Web forms with a hidden variable containing the session identifier. Since sessions are started with a reactive pull and continued with session-oriented pulls it is not needed to use rewritten URLs within one session: pulls always return a Web form. Each instance contains a variable in which the collection of its active sessions is stored.

A result of this way of implementing sessions is that parallel sessions within one instance are automatically implemented. However, we do have to take care that parallel sessions do not interfere while accessing instance dependent data. This is prevented by synchronizing data access.

Each time a servlet is called it checks whether a session identifier is passed. If so, it checks whether the session occurs in the corresponding instance and continues the session if that is possible. If the session cannot be found or if no session identifier is passed, a new session is started.

4. EXAMPLE: THE GIFT SELECTION

In this section we give an example of a way to distribute gifts over invitees for a marriage. We specify an Internet application via which this distribution takes place. We do not give a full specification. Instead, we give parts of the specification which will be sufficient to get an idea of the way in which the different part of the application are specified.

First of all, we have to specify which roles are applicable to the problem. An initiator must specify which gifts can be given away and who are invited for the marriage. The invitees must be able to select a gift they want to donate to the bridal couple. This means that we have two roles: *Initiator* and *Invitee*.

```
role
  Initiator;
  Invitee;
end role
```

Next, we specify which variables and functions we make use of. We need a variable to refer to the initiator and one to refer to the invitees. Furthermore, the gifts (of type String) are stored in a variable. We make use of a deadline before which the invitees have to select the gift they want to give.

```
var initiator: Initiator;
var invitees: set of Invitee;
var gifts: set of String;
var deadline: Deadline;
```

Functions are also specified in the data part. The bodies of the functions are specified in the Java language. We can use the variables specified before. Variables which represent a set are implemented as objects of the Java *Vector* class. Therefore, we can make use of the methods of this class in the bodies of the functions. Some examples of functions we have specified are given below.

```
function process_selection(gifts: set of String,
      gift: String): String
  = java
      if (!gift.equals("") &&
          (gifts.indexOf(gift)>-1)) {
        gifts.removeElement(gift);
        return "yes";
      } else
        return "no";
    end java;

function gifts_left(gifts: set of String): Boolean
  = java
      return !gifts.isEmpty();
    end java;
```

Next, we specify the Web pages/forms and e-mails we use to interact with the users. We declare the kind of each interaction and the role a user must have to interact. A Web page/form is specified by its title and body, an e-mail by its sender, receiver, subject and contents. We use plain text and references to input/output parameters. A Web form which we use to ask the initiator to insert a deadline is given below. In the interaction, a regular expression is added to check the syntax of the text which is typed out in the input field. If the text does not answer the syntax, a message is shown and the text must be altered until it does satisfy the syntax.

```
session of Initiator → set_deadline(
      in deadline: Deadline) =
  { title:
      text: "Gift selection";
    body:
      text: "Insert deadline (dd-mm-yyyy hh:mm:ss):";
      input: deadline
             check "/^\d\d-\d\d-\d\d\d\d
                    \d\d:\d\d:\d\d$/"
             else "Incorrect date format.";
  };
```

A specification of an e-mail is given below. The e-mail is sent to each invitee. He is asked to visit the application's URL, log in and select a gift. A "\n" specifies a line break.

```
mail to Invitee ← invitation_email(
      out initiator: Initiator, out deadline: Deadline,
      out invitee: Invitee, out gifts: set of String) =
  { from:
      output: initiator.email;
    to:
      output: invitee.email;
    subject:
      text: "Invitation for gift selection.";
    contents:
      text: "Hello ";
      output: invitee.name;
      text: ",\n\nYou are invited to select a gift.
             \n\nVisit the following url:\n\n";
      output: URL;
      text: "\n\nThe gifts are:\n";
      output: gifts;
      text: "\n\nDeadline before which you have to
             select your gifts:\n";
      output: deadline;
      text: "\n\nUse the following name and password
             to log in:\nName:";
      output: invitee.name;
      text: "\nPassword: ";
      output: invitee.password;
      text: "\n\ngreetings, ";
      output: initiator.name;
  };
```

After specifying all interactions, we have to specify the different sessions. Each session has a name. A session is specified by a sequence of (inter)actions. We have to specify two sessions.

First of all, we specify the *initialization* session. In this session the initiator is asked to insert all relevant data which is needed for the gift selection, i.e. his name and e-mail address, the set of gifts, the set of invitees and the deadline before which the sessions with the invitees must take place.

```
initialization =
  { start session of initiator → set_initiator(initiator);
    while incorrect_deadline(deadline) do
      session of initiator → set_deadline(deadline);
    od;
    s = yes();
    while equals_yes(s) do
      session of initiator →
          add_invitee(invitees, invitee, s);
      process_invitee(invitees, invitee);
    od;
    s = yes();
    while equals_yes(s) do
      session of initiator → add_gift(gifts, gift, s);
      process_gift(gifts, gift);
    od;
    for all j ∈ invitees do
      mail to j ←
          invitation_email(initiator, deadline, j, gifts);
    od;
    end session of initiator ← thank_you_initiator();
  };
```

Furthermore, we specify the *selection* session. In such a session an invitee is asked to select a gift. First, the invitee has to log in using his name and password (this is indicated by the attribute *authenticate from*). After selecting a gift, a check is done. If the gift is still available it is removed from the set of available gifts and attributed to the invitee. If it has been attributed to another invitee a new gift must be selected.

```
selection =
  { start session of invitee →
        authenticate from invitees;
    session of invitee → select_gift(gifts, gift);
    s = process_selection(gifts, gift);
    while equals_no(s) do
      session of invitee → again_select_gift(gifts, gift);
      s = process_selection(gifts, gift);
    od;
    end session of invitee ← thank_you_invitee(gift);
  };
```

Finally, we have to specify in which order the sessions must take place. The application starts with the *initialization* session. After that, *selection* sessions can take place as long as the deadline has not been reached and gifts are available for distribution.

```
application

  session initialization;
  until deadline do
    while gifts_left(gifts) do
      session selection;
    od;
  od;

end application
```

This example has been implemented and can be executed as a Java Servlet. The *while* construction which is used in the final part of the specification is implemented as a parallel composition. This means that a number of *selection* sessions can be executed in parallel. Since it is possible to select a gift which, in the meantime, has been selected by another invitee in a parallel session, we have added the check to the *selection* session.

5. RELATED WORK

We introduced a specification language for a specific class of Internet applications, viz. applications for distributed consensus. There are many different languages to specify Internet applications, but as far as we know, none of them is specifically designed to develop such applications. We will discuss some of them and show in what way they agree with or differ from *DiCons*.

Closest to our work is the development of the Web-language *Mawl*, [1, 19]. This is also a language that supports interaction between an application and a single user, and adds a state concept to HTML. Mawl provides the control flow of a single session, but does not provide control flow across several sessions (the only thing that persists across sessions are the values of global variables). This is a distinguishing feature of *DiCons*: interactions involving several users are supported. On the other hand, Mawl does allow several sessions with a single user to exist in parallel, using an atomicity concept to execute sequences of actions as a single action. Mawl does not use Java servlets.

Groupware is a technology designed to facilitate the work of groups. This technology may be used to communicate, cooperate, coordinate, solve problems, compete, or negotiate. Groupware can be divided into two main classes: asynchronous and synchronous groupware. Synchronous groupware concerns an exchange of information, which is transmitted and presented to the users instantaneously by using computers. An example of synchronous groupware is chatting via the Internet. On the other hand, asynchronous groupware is based on sending messages which do not have to be read and replied to immediately. Examples of asynchronous groupware that can be specified in *DiCons* are work-flow systems to route documents through an office and group calendars for scheduling projects. More information on groupware can be found in [27].

Visual Obliq [3] is an environment for designing, programming and running distributed, multi-user GUI applications. Its interface builder outputs code in an interpreted language called *Obliq* [5]. Unlike *DiCons* applications, Obliq applications do not have to run on one single server: an application can be distributed over several so-called sites. After setting up a connection, sites can communicate directly. In this way, an application can be partitioned over different servers. Another difference with respect to *DiCons* is that a client has to install a special interpreter to view Visual Obliq applications whereas *DiCons* makes use of standard client-side techniques like HTML pages which can be viewed using a Web browser. In [4], embedding distributed application in a hypermedia setting is discussed and in particular how applications generated in the Visual Obliq programming environment are integrated with the World Wide Web. Here,

a Web browser is used to refer to a Visual Obliq application, but it must still be viewed using an interpreter.

Collaborative Objects Coordination Architecture (COCA) [21] is a generic framework for developing collaborative systems. In COCA, participants are divided into different roles, having different rights like in *DiCons*. Li and Muntz [20] used this tool to build an online auction. A COCA Virtual Machine runs at each client site to control the interactions between the different clients. On the other hand, any client connected to the Internet can communicate with a *DiCons* application without having to reconfigure his machine.

The *Describing Collaborative Work Programming Language (DCWPL)* [7] helps programmers to develop customizable groupware applications. DCWPL does not concern the computational part of an application. As in *DiCons*, this part is specified in a computational language like Java, Pascal or C^{++}. A DCWPL application also runs on an interpreter, here called control engine. DCWPL is based on synchronous groupware in contrast to *DiCons* in which the asynchronous aspect is more important.

Further, there are languages that allow to program browsing behaviour. These, for instance, allow to program the behaviour of a user who wants to download a file from one of several mirror sites. For so-called *Service Combinators* see [6, 18]. A further development is the so-called *ShopBot*, see [8].

Our implementation is based on existing Internet programming techniques, viz. Java servlets and HTML. In Udell's book on groupware [27] an Internet vote is implemented using a Java servlet. Also in [24] an election servlet is presented. Furthermore, there are commercial voting servlets put on the market. One of them can be found at [9]. To set up an Internet auction one can use commercial software like *rAuction*, which can be found at [26].

Other useful Internet programming techniques are Active Server Pages (ASP) [15] and Java Server Pages (JSP) [13]. We can extend these techniques with customized tags for distributed consensus. However, these techniques are library-based and therefore not as suitable for formal verification as our language-based *DiCons* technique.

6. CONCLUSIONS

We designed a language that supports the development of Internet applications at the right level of abstraction. Although we have done several experiments with the language, we plan to gain more experience, by using the language for larger applications. This will probably show options for refining and extending *DiCons*. Sample specifications of a voting system, an auction system and a Meeting Scheduler already indicated some useful extensions. We mention: atomic regions (to support mutual exclusion, as in Mawl [1, 19]), database coupling (for processing information available at the system, as in Strudel [10]), and style sheets (to give the Web forms a more professional appearance).

Since the communications with the users of the system are under dynamic control, based on the system state, *DiCons* supports personalized and adaptive interactions.

Our choice to base *DiCons* and its support tools on existing and readily available Internet technology, makes it very easy to use. Nevertheless, the language and tools can be easily extended to support more advanced communication schemes.

One of the motivations for designing *DiCons* was that it would allow for the development of formally verified Internet applications. Therefore we prefer a language-based approach to a library-based approach. Up to now, we have not gained experience with formal verification of *DiCons* programs. Current research is focussed on finalizing the formal semantics for the behavioral part of *DiCons* and to experience with formal validation based on this semantics.

We implemented a compiler to compile *DiCons* specifications into Java Servlets. Except for generating a Servlet, the compiler checks a specification on its syntax and static semantics.

More information on *DiCons*, its compiler and some working examples can be found in [2] or at *http://pc32.eesi.tue.nl/*.

7. REFERENCES

[1] D. L. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: A domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346, May/June 1999. Special Section: Domain-Specific Languages (DSL).

[2] H. v. Beek. Internet protocols for distributed consensus – the *DiCons* language. Master's thesis, Eindhoven University of Technology, Aug. 2000.

[3] K. Bharat and M. H. Brown. Building distributed, multi-user applications by direct manipulation. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Groupware and 3D Tools, pages 71–81, 1994.

[4] K. Bharat and L. Cardelli. Distributed applications in a multimedia setting. In *Proceedings of the First International Workshop on Hypermedia Design*, pages 185–192, Montpellier, France, 1995.

[5] L. Cardelli. Obliq A language with distributed scope. SRC Research Report 122, Digital Equipment, June 1994.

[6] L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, May/June 1999.

[7] M. Cortes and P. Mishra. DCWPL: A programming language for describing collaborative work. In *Proceedings of ACM CSCW'96 Conference on Computer-Supported Cooperative Work*, Language Support for Groupware, pages 21–29, 1996.

[8] R. B. Doorenbos, O. Etzioni, and D. S. Weld. A scalable comparison-shopping agent for the world-wide web. In W. L. Johnson and B. Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 39–48, Marina del Rey, CA, USA, 1997. ACM Press.

[9] Virtua – fastvote support, 1997–1999. http://www.virtua.com/fastvote/, Virtua Communications Corporation.

[10] M. Fernández, D. Suciu, and I. Tatarinov. Declarative specification of data-intensive Web sites. *ACM SIGPLAN Notices*, 35(1):135–148, 2000.

[11] J. Friedl and A. Oram. *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools (O'Reilly Nutshell)*. O'Reilly & Associates, Inc., first edition, Jan. 1997.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series. Addison-Wesley, second edition, June 2000.

[13] M. Hall. *Core Servlets and JavaServer Pages*. Sun Microsystems Press/Prentic Hall PTR, June 2000.

[14] N. Heinle and R. Koman. *Designing with JavaScript*. O'Reilly & Associates, Inc., May 2000.

[15] A. Homer, D. Sussman, and B. Francis. *Professional Active Server Pages 3.0*. Wrox Press Inc, Sept. 1999.

[16] J. Hunter and W. Crawford. *Java Servlet Programming*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1998.

[17] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1997.

[18] T. Kistler and H. Marais. WebL — a programming language for the Web. *Computer Networks and ISDN Systems*, 30(1–7):259–270, Apr. 1998.

[19] D. Ladd and J. Ramming. Programming the web: An application-oriented language for hypermedia service programming. In *Proc. 4th WWW Conf., WWW Consortium*, pages 567–586, 1995.

[20] D. Li and R. Muntz. Building online auctions from the perspective of coca. *Submitted to HICSS-33*, Jan. 2000.

[21] D. Li and R. R. Muntz. COCA: Collaborative objects coordination architecture. In *Proceedings of ACM CSCW'98 Conference on Computer-Supported Cooperative Work*, Infrastructures for Collaboration, pages 179–188, 1998.

[22] S. Mauw, M. Reniers, and T. Willemse. Message Sequence Charts in the software engineering process. In *Handbook of Software Engineering and Knowledge Engineering, S.K. Chang, editor*. World Scientific, 2001. To appear.

[23] Metamata home page: Javacc documentation. http://www.metamata.com/JavaCC/docs/, Fremont, California.

[24] L. O'Brien. Vox populi. *Java Pro Magazine*, June 1999.

[25] D. Raggett, A. L. Hors, and I. Jacobs. Html 4.01 specification. Technical report, W3C User Interface Domain Recommendation, Dec. 1999.

[26] Siteoption home page. http://www.siteoption.com/, SiteOption.com, Green Cove Springs, USA.

[27] J. Udell. *Practical Internet Groupware*. O'Reilly & Associates, Inc., Oct. 1999.

[28] A. Williamson. *Special Edition Using Java Servlet API*. Que Corporation, Indianapolis, IN, USA, 1997.

8. BIOGRAPHY

Jos Baeten is full professor in computing science at Eindhoven University of Technology, The Netherlands. His specialisation area is formal methods, in particular concurrency theory, formal specification languages and term rewrite systems. Especially, he is known for his work in process algebra. He has a Ph.D. in mathematical logic from the University of Minnesota in 1985, and since worked at Delft University of Technology, the University of Amsterdam and the Centre for Mathematics and Computer Science (CWI) in Amsterdam, before moving to Eindhoven in 1991.

Harm van Beek is a Ph.D. student at the Formal Methods Group of the Eindhoven University of Technology, where he also received his master's degree in computing science (cum laude, 2000). He is working on the formal development of Internet applications at the Eindhoven Embedded Systems Institute (EESI) which is an initiative of the faculties of Mathematics & Computing Science and Electrical Engineering of the Eindhoven University of Technology.

Sjouke Mauw is associate professor at the Eindhoven University of Technology and senior researcher at CWI (Center for Mathematics and Computer science) in Amsterdam. He received his master's degree in mathematics in 1985 from the University of Amsterdam, where he also obtained his Ph.D. degree in computer science (1991). He graduated on the Ph.D. thesis "PSF - A process specification formalism" which contains the design of a specification language based on process algebra and abstract data types. His research concerns the theory and application of formal methods for specification and verification of concurrent systems. He was associate rapporteur for study group 10/Q9 of the ITU (International Telecommunication Union), where he supervised the standardization of the semantics of Message Sequence Charts.