An MSC Based Representation of *DiCons*

J.C.M. Baeten, H.M.A. van Beek, and S. Mauw

Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands.
{josb,harm,sjouke}@win.tue.nl

Abstract. We present a graphical MSC-based representation of the language *DiCons*, which is a formal language for the description of Internet applications.

1 Introduction

Building internet applications is not an easy task. Given the many problems involved it makes sense to investigate the use of formal methods since we think that formal methods can help to develop Internet applications more efficiently, and can help to improve the quality of applications.

Currently, a mix of different languages, at different levels, with a low degree of formality is used, e.g. Perl, C++ and Java. Recently, we have started a new line of research in order to remedy this. This has resulted in the first version of the language *DiCons* in [3] of which an extended abstract appeared as [2].

The most important feature of *DiCons* is that it is geared towards the highest level of abstraction, the communication level, and that aspects of lower levels are treated in separate parts of the language. The purpose of this paper is to give a graphical presentation of *DiCons* specifications.

The language *DiCons* focuses on a specific class of internet applications, a class we call Distributed Consensus (this explains the name of the language). This is the class of applications where several users strive to reach a common goal without having to meet face to face, nor will there be any synchronized communications between users. A central system, viz. an Internet application, must be used to collect and distribute all relevant information. Example applications are making an appointment, evaluating a paper, and selecting a winner.

Currently, we are working on the formal semantics of the language, which serves as the starting point for this paper. The papers [3, 2] show the usefulness of the language in a number of examples which were first developed as MSC scenarios, and afterwards programmed in *DiCons*. MSC is the language Message Sequence Chart, that is used a lot in the telecommunications industry, as standardized in [11]. MSC has in common with *DiCons* that it is mainly concerned with the interaction between system components (here: a server and several clients) and that internal processing of information is less important.

A closer look reveals that drawing the MSCs does not leave out much information. Mostly, they contain just one scenario (or a couple of related scenarios).

On the other hand, the examples in *DiCons* suggest that there is a main trace that admits variations occasionally. With the recent extensions of MSC [11] it could be possible to give a complete, or almost complete MSC specification of *DiCons* programs. This is the hypothesis that we investigate in this article.

There are several reasons why an MSC-like representation can have added value for a textual specification language like *DiCons*. Most important is that a visual interface can aid communication with a customer, who wants an application to be built. It is easier to understand by those not used to programming languages. Focusing on example scenarios can be very important in the initial design phase of a new application.

On the other hand, there are reasons why just using MSC as a trace description language is not enough. Most important, just describing traces can leave ambiguities, obscurities and misunderstanding about the working of an application.

In general, it is not advisable to give complete system specifications in MSC. However, it is interesting to note that *DiCons* is intended for restricted class of applications, and this makes it possible to define a complete MSC-like representation. Thus, in *DiCons* we only consider the behaviour of the server, depending on possible external stimuli and the internal state. This means we only have to define the complete behaviour of a single MSC instance. This appears to be a lot simpler than specifying the complete behaviour of several instances.

In this paper, we investigate giving an MSC-like representation of *DiCons* with comparable expressivity. A first try is to see whether MSC-2000 is powerful enough by itself, but it soon turns out that more is needed. For instance, *DiCons* involves several communication primitives that have to be represented in different ways. We see that *DiCons* has compound communications that go beyond the simple scheme of an asynchronous MSC communication. This requires extensions of MSC-2000 in order to raise the representation to the same level of abstraction. Our extensions are in the style of MSC-2000, for instance regarding the use of in-line expressions.

We are not in the business of proposing extensions of the language MSC. Rather, we look upon this work as a special application of MSC. In our experience, every (new) application domain of MSC will lead to a comparable adaptation of the language. This is due to the nature of the language MSC. On the one hand, MSC is so universal as to be applicable whenever there is a form of distribution and communication. On the other hand, the drive to express issues in the appropriate way and at the appropriate level will necessitate new features that have not been standardized by the ITU (yet). The present offering of MSC-2000 seems to have enough features already. There is a good basis of possibilities to express issues like modularization, data, time, and many more things, and it is not obvious that specific applications should lead to even more extensions of the language.

Rather, we see our work as defining a graphical layer on top of *DiCons* based on MSC, and not as an extension of MSC. This paper is exploratory: we do not give a formal graphical syntax and do not give a translation to the semantics.

On the other hand, we have tried to have a one-to-one correspondence with the semantical constructs.

Also, it is not our intention to use exactly the semantics of MSC. For example, we introduce several communication primitives that do not allow a simple reduction to existing primitives. Thus, the semantics of our MSC-like language will not arise by translation to the semantics of MSC, but rather by a translation to the abstract syntax of *DiCons* and from there to the semantics of *DiCons*. As a side remark, the semantics of MSC and the one of *DiCons* are not so different, since both are based on a translation to process algebra.

In the following section, we present a short introduction to the *DiCons* language and define our graphical representations for the language primitives. In the next section, we work out an example, the meeting scheduler. Finally, we present some conclusions.

2 *DiCons* Primitives and their Graphical Representation

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*.

In order not to 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 semantic 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.

2.1 Overview of Language Constructs

Keeping the above considerations with respect to the application domain and available technology in mind, we come to a description of the basic constructs

of *DiCons*. Although *DiCons* is initially developed as a linear language we will only give a description of the graphical representation of *DiCons*. To this end we build on the work that is done on the development of the MSC standard. A graphical *DiCons* specification looks like an MSC, however, it *is not* an MSC. We will define some extensions to the MSC standard that are essential to reach the level of abstraction of the current linear *DiCons* language in a natural way. One can easily see whether a figure is a basic MSC or its *DiCons* version: keyword **msc** in the upper left corner is replaced with keyword **DiCons** in the *DiCons* version.

Apart from basic MSCs we will also make use of High-level MSCs (see [16]) to specify *DiCons* applications. In the *DiCons* version we extend HMSCs by adding constructs for declaring roles and variables.

We will first list the language ingredients and later discuss these in more detail, by defining their graphical syntax. We will not give a complete description of the syntax and semantics of (graphical) *DiCons*. The example in Sect. 3 will serve to show the flavor of the graphical *DiCons* syntax and the way in which the language can be used.

central application The central application is the main part of our *DiCons* language. All interactions take place via this application.

users and roles 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 presented to the user by concrete means of communication, 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.)

2.2 Central Application

Since all interactions take place between the central system and one of the users, the central system must be included in all graphical *DiCons* specifications. We can only give a decription of the behaviour of the central system. We cannot *force* users to interact in the way we intend, therefore we assume that they will do so in order to be able to give a useful specification. The application is represented by a gray-headed instance in a wide form not having an instance name. In Fig. 1 a graphical representation of the central application is given.

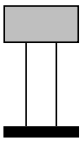

Fig. 1. The central application

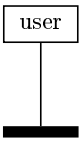

Fig. 2. A user

2.3 Users and Roles

A user is a (possibly human) entity that can interact with the system. 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.

In the graphical *DiCons* syntax we define roles by introducing them in the same way as we introduce variables (see Sect. 2.5). However, we use the **role** keyword instead of the **var** keyword. We represent a user by a regular instance, containing the name of the user in its head symbol (Fig. 2).

2.4 Interactions

The basic problem when defining the interaction primitives is to determine the right level of abstraction. In order to get a feeling of the level of abstraction which is optimally suitable, look at Fig. 3. In this drawing we sketch in MSC a typical scenario of an Internet application which is called the *Meeting Scheduler* (see [17]). 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.

Please note that this is not a graphical *DiCons* specification, but an ordinary MSC describing a possible scenario. We will give a more detailed graphical *DiCons* specification in Sect. 3.

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 do not know that they will be invited for a meeting, the initiative of this interaction is at the server

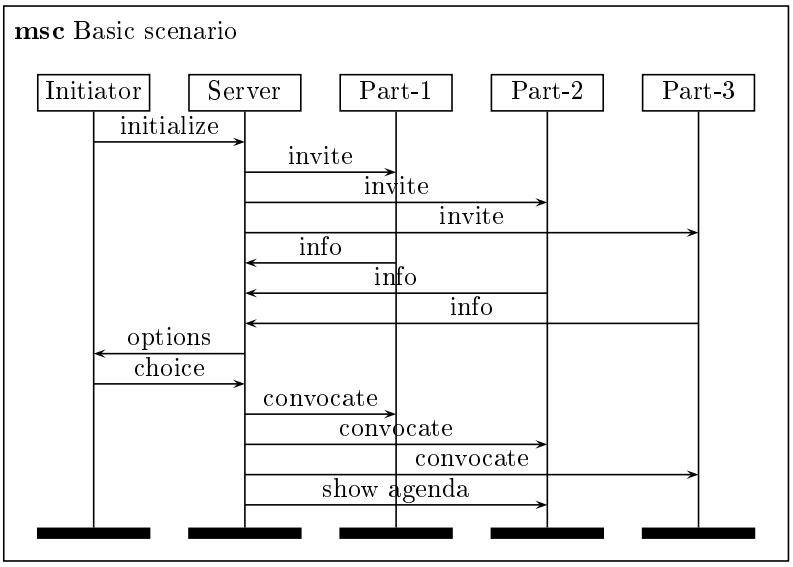

Fig. 3. An MSC Scenario of an Internet Application

side. The way in which a server can actively inform a client is (for example) by sending an e-mail. This interaction only contains information transmitted from the server to the user. The messages *options* and *convocate* can also be 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 Fig. 4. 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. So a session is composed of three kinds of interactions.

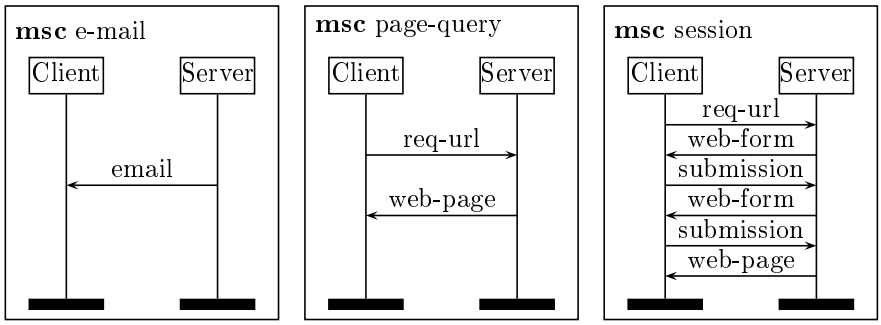

Fig. 4. Interaction Primitives

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 direction of the arrow indicates whether the interaction involves a push or a pull.

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. To graphically indicate the initiating party we place half a circle between its instance and the arrow representing the interaction. The filling of the circle indicates whether the interaction takes part in a session or not.

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. We make use of $\triangleright$ and $\triangleleft$ to graphically specify the direction in which data flows. Parameters left to a $\triangleright$ flow from left to right and parameters right to a $\triangleleft$ flow in the opposite direction.

The notation for our communication primitives is given below. We give a basic MSC to show in which order the different messages take place. Furthermore, we give the corresponding graphical *DiCons* syntax. In the figures given below, i_k $(0 \leq k \leq m)$ denotes an input parameter and o_k $(0 \leq k \leq n)$ an output parameter.

active server push The server takes the initiative to send information:

An *active push* takes place if the server sends a message to a client which is not directly the result of a request from that client. Such an interaction

can only take place via an e-mail and not by sending a Web page, since this requires a client to request some URL.
We denote this interaction by a message from the server to the client (Fig. 5). The circle at the server side means that the server initiates the interaction. It is not filled since the interaction does not take part in a session. The direction of the arrow indicates that we have to do with a push.

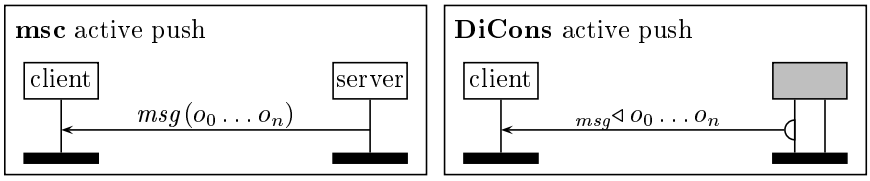

Fig. 5. Active push

reactive server push The server sends a Web page on request of the client:
A *reactive push* takes place if the server sends a Web page, not containing a Web form, to a client which is the result of a normal request from that client, (that is, not generated by filling out a previously received Web form). Here, the circle is placed at the client side, meaning that the client initiates the interaction. Again, it is not filled since the interaction is not part of a session. Actually, the interaction may be seen as one that both starts and ends a session containing only this interaction.

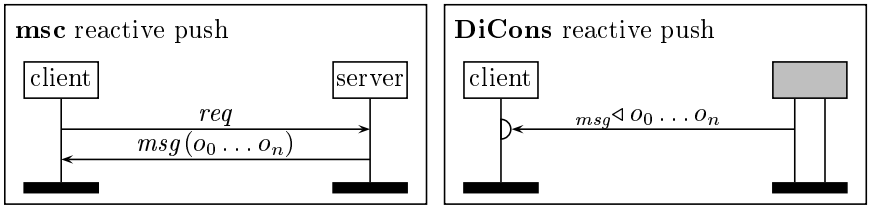

Fig. 6. Reactive push

reactive server pull This interaction takes place if a client sends a request to the server on which the server responds by sending a Web form. This form is filled in and submitted by the client. A reactive pull starts a session with one particular client. The client starts the interaction so the circle is at the client side. The upper half of the circle is not filled since no session existed prior to this interaction. Its lower half is filled, which means that after ending this interaction a session is open. Note that the direction of the arrow indicates a server pull.

The dashed line in the MSC in Fig. 7 is *not* part of this interaction primitive. It means that this interaction must be followed by an interaction which is started by sending a message from the server to the client.

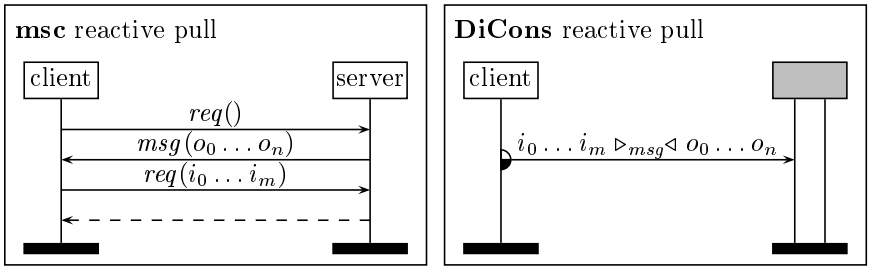

Fig. 7. Reactive pull

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.

The server initates this interaction and therefore the circle is at the server side (Fig. 8). It is completely filled, since a session existed at the beginning of the interaction and is still open at the end.

Again, the dashed lines represent mandatory messages preceding and following this interaction. Note that both a reactive pull and a session-oriented pull can precede this interaction and that a session-oriented pull or a session-oriented push can follow it.

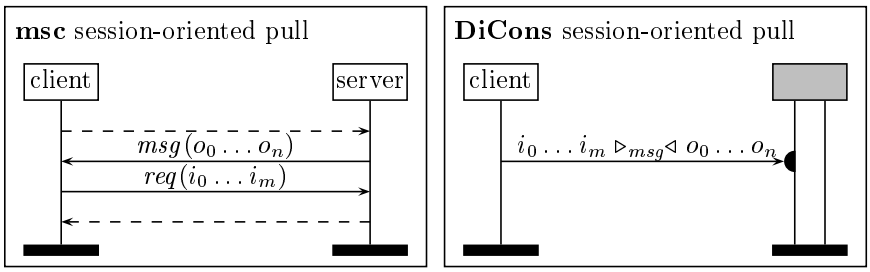

Fig. 8. Session-oriented pull

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.

If a server/client communication takes part in a session, the server can send a Web page, not containing a Web form, as a response on a submission of a Web form. Such a *session-oriented push* ends the session because the client can no longer fill out any forms. The server initiates this interaction so the circle is at the server side (Fig. 9). The upper half of the circle is filled, meaning that the interaction starts within a session. On the other hand, the lower half is not filled which denotes that the session is no longer open after ending this interaction.
The dashed line in the MSC indicates a message from the client to the server, which must precede this interaction.

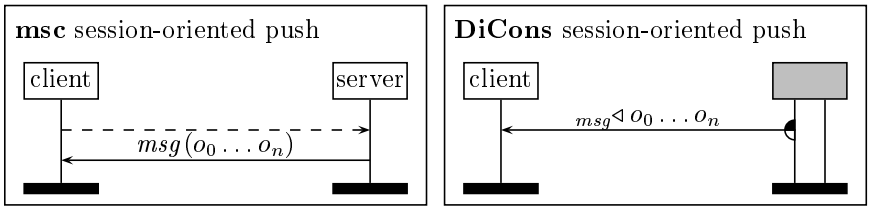

Fig. 9. Session-oriented push

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 Behaviour

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*. 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*. 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.

sequential composition We make use of the basic MSC based representation for sequential composition. We only specify the behaviour of the central application, so the order of the events on the instance axis of this application determines their causal connection.

conditional branching For the construct of a conditional branching we make use of an inline expression having the keyword **if** followed by the condition in its upper-left corner. See Fig. 10 for an example of the grapical syntax for expression *if b then X else Y fi*.

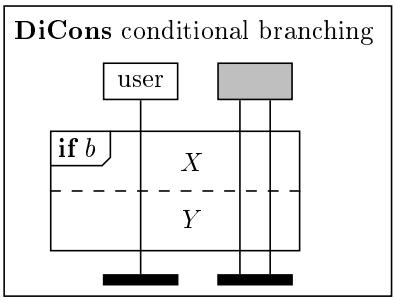

Fig. 10. Syntax for conditional branching

repetition We make use of two different statements for repetition. First of all we have the *for all* $s{\in}S$ *do* $X(s)$ *od* statement. For all elements s in S, bodies $X(s)$ are sequentially executed in an arbitrary order, where S represents a finite set of data elements. Furthermore, we can specify a while loop by using the *while* b *do* X *od* statement. The while loop repeats statement X until test b proves false. In Fig. 11 the syntax for both loops is given. The X represents a collection of (inter)actions. If variable s occurs in X we indicate this by writing $X(s)$.

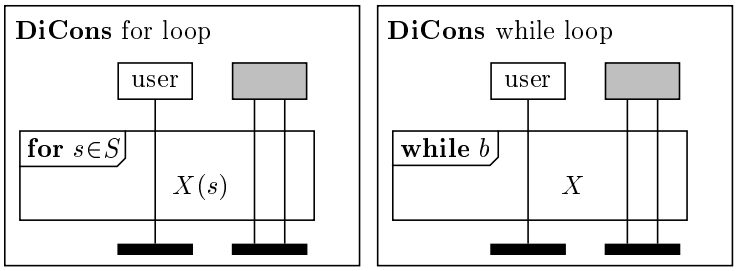

Fig. 11. Syntax for repetition

parallel composition We also have two operators for parallel composition at our disposal: the *fork* and the *par* operator (Fig. 12).

Statement *fork* $u{\in}U$ *do* X *od* means that all users u ($u \in U$) can execute (inter)actions X between user u and the central applications in parallel and more than once. On the other hand, the *par* $u{\in}U$ *do* X *od* statement specifies that all users u ($u \in U$) will execute (inter)actions X between user u and the central applications in parallel but only once. So, in contrast to the *fork* operation, after execution of X for all $u \in U$ the *par* operation ends.

A *fork* gives clients the possibility to start an interaction (for example via a reactive push) while the *par* obliges a client to interact. This obligation only

makes sense if the initiative for the interaction is on the server-side, such as an active server-push.
Note that both the instance head and the instance foot of u are placed inside the operator's frame. This means that instance name u is bounded by the operator and therefore it is not known outside the frame.

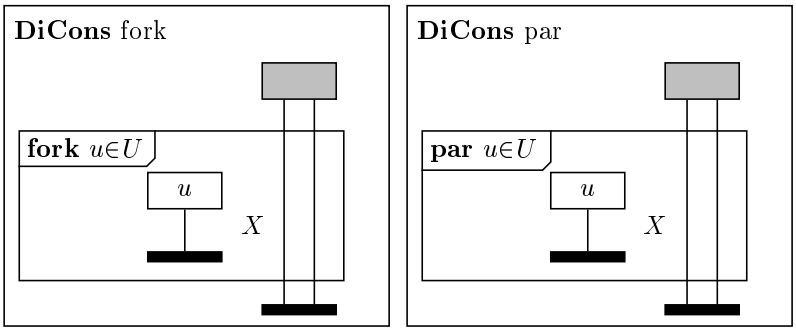

Fig. 12. Syntax for parallel composition

time-outs using conditional disrupts We introduce the *until b do X od* statement to specify conditional disrupts. This means that X is normally executed until b becomes true. At that moment the statement ends, independent of the (inter)actions that are taking place at that moment. If X ends before b becomes true the statement ends too. By placing a time check in b we can specify time-out interrupts. However, b may be an arbitrary boolean expression containing predicates on any part of the state space.

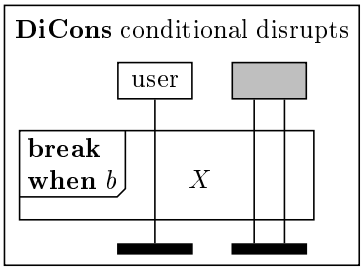

Fig. 13. Syntax for conditional disrupts

variable assignments and procedure calls In the textual syntax of *DiCons* we have a *data* part to introduce types, variables and functions. We can also introduce them in several ways in the graphical representation. First of all, we can introduce variables directly below the **DiCons** keyword (Fig. 14).

These variables are available to all elements in the MSC, so the frame surrounding the instances defines the block in which these variables are known. As mentioned in Sect. 2.3, we can introduce roles in the same way by using keyword **role**.

Furthermore, we can introduce variables using an inline expression. The variables are only available within the box that is used for this variable introduction. If we want to assign an intial value to a variable we can do this at the place of declaration using the ":=" sign, however, this is optional.

Variables are owned by the central application only. This means that problems concerning individual variables in basic MSCs [10, 9] do not arise in our graphical *DiCons* syntax.

There are three ways to change the value of a variable:

- Via a local action of the server, i.e. an assignment or a function call;
- Via an input parameter of an interaction;
- Via the bind-construct in the header of a graphical *DiCons* specification.

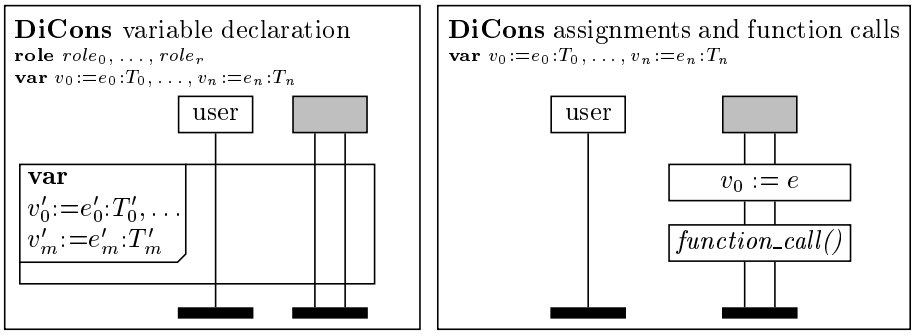

Fig. 14. Syntax for variable declarations, variable assignments and function calls

High-level behavioural composition To be able to compose the behaviour description of a *DiCons* application in an hierarchical way we make use of the same notation as is used for the composition of MSCs by means of High-level MSCs (see [16]).

2.6 Presentations

Up to now, we only described the top level view of *DiCons* applications. This level is concerned with the composition of interactions into complete specifications of the behaviour of the applications. However, this is not the only level for which we would like to make a graphical representation. The other levels of the *DiCons* language concern the presentation of the interactions and the data definition. Examples of textual descriptions of these levels can be found in [3].

3 An Example: The Meeting Scheduler

The purpose of this section is to explain the use of graphical *DiCons* by means of an example. The example concerns the specification of a Meeting Scheduler, taken from [17]. Figure 3 in Sect. 2.4 already contains an informal example of a scenario of this application.

The purpose of a Meeting Scheduler is to support the process of scheduling a date for a meeting without the intended participants having to meet for selecting a date. A central server could easily take care of the administrative tasks and support the communication process. Of course there are several publicly available tools with capabilities similar to our Meeting Scheduler which offer much more functionality.

We present a top-down development of the case study. The highest level is represented in Fig. 15. This contains a drawing which resembles a High-level Message Sequence Chart (see [16]). It shows that the application consists of three phases, which are called *initialize*, *check*, and *select*. These three phases are elaborated in Figs. 16, 17 and 18. Roughly speaking, in the initialization phase the initiator starts the application and provides the server with the required information, in the check phase, the intended participants send in their selection of suitable dates, and in the select phase the initiator selects the meeting date. In these three phases, several global variables may be used. These are declared just below the header. Since we will not focus on the actual language used for defining these variables and other data-related objects, we will simply use an abstract mathematical notation. Users who interact with the system can have two roles, viz. initiator and invitee. Furthermore, we declare an initiator of type Initiator, a set of Invitees in which the list of invitees is stored, a set of optional dates, a deadline before which the intended participants must have replied, and for each intended invitee the list of dates that are not convenient to him.

Next, we look at the specification of the initialization phase in Fig. 16. This is a specification at the interaction level. There are three entities taking part in this description. The middle entity describes the system providing the required service. Its behavior is represented by a, so-called, fat instance axis. This is to clearly distinguish it from the other instances acting as clients. The left instance is named *initiator*. This is the user who initializes the application. Because any user is allowed to initialize the application, we add the **bind** construct in the header to indicate that the identity of the actual initiator is saved in the variable named *initiator*. From now on, this variable is instantiated. The rightmost entity has the name *i*. This is a variable local to the par-frame.

The first interaction, named *inv*, implies a flow of information from the initiator to the server which is stored in the variable with the name *invitees*. Of course the initiator must send data of the intended type (a set of invitees) in order to successfully complete this interaction. This can be enforced by using a web form and JavaScript, but we will not elaborate on these implementation related issues. Note that the interaction is a reactive server pull (see Sect. 2.4), specifying the beginning of a session. The second interaction within this session, a session-oriented pull, requires the initiator to send a list of possible dates, fol-

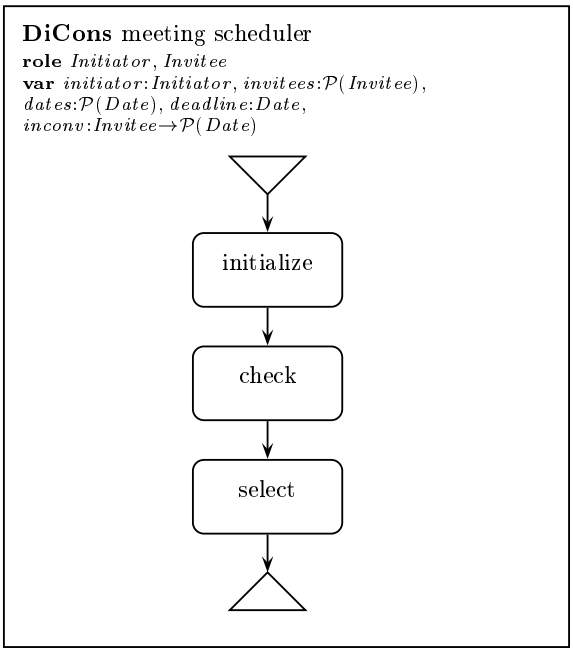

Fig. 15. High-level overview

lowed by an interaction which requires the initiator to send the deadline before which the intended participants must have replied.

After having received all this information, the server invites all users from the set of invitees to submit their selection of inconvenient dates. This can be implemented by providing them with a URL of some web page where each of them can select some of the proposed dates. The fact that all invitees are informed in this way is expressed in the upper left corner of the surrounding frame, which contains the *parallel* operator. Finally, the server ends the session by sending a confirmation to the initiator.

In the second phase (Fig. 17), all intended participants are allowed to submit their lists of inconvenient dates any number of times. This can be done until all participants have replied, or the given deadline for replying has passed. This is indicated by the **break**-frame. The contained behavior will be disrupted at the moment that the breaking condition becomes true. Please note that this frame in no way indicates any repetition itself. It just expresses that the enclosed behavior can be disrupted. The repetition is expressed in the **fork**-frame. This fork operator makes it possible for every invitee i from the set of invitees to start a session. It is allowed that these sessions run in parallel and that an invitee takes part in more than one session. It is even possible for one invitee to run several independent sessions in parallel. For each session within this **fork**-frame the server declares a local variable *ds*, which will contain the list of inconvenient

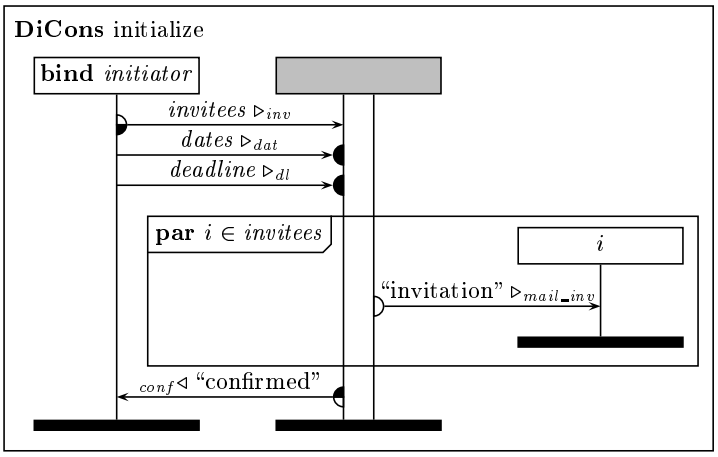

Fig. 16. Initialization phase

dates as submitted by the invitee in this session. A **var**-frame is used to indicate the scope of this variable. An invitee starting such a session is prompted by the server with the list of optional dates and the list of inconvenient dates possibly provided by i in a previous session. In the same interaction i provides his (new) list of inconvenient dates *ds*. The assignment in the local action adds this list to the information maintained by the server, and the server ends the session by sending a confirmation to the invitee. Finally, the server adds the name of the invitee to the set of invitees *checked*. This is to keep track of which invitees have already submitted their information. This variable is used in the guard of the **break**-frame. If this guard becomes true, the fork will be disrupted and the server concludes that enough information has been collected in order for the initiator to select the most suitable date. This is indicated to the initiator in the final interaction of this drawing.

In the last drawing (see Fig. 18) the initiator starts a session to select the date for the meeting and notify the participants of this date. In the first interaction the server provides the initiator with the list of inconvenient dates. Based on this information the initiator decides upon the final date and sends it to the server, where it is stored in local variable *d*. In a **par**-frame, the server sends an invitation or notification to all invitees, making a distinction between invitees that have indicated to be able to come on the selected date and those that cannot come. Finally, the server sends a confirmation to the initiator and the application ends.

4 Related Work

We introduced a graphical representation of a specification language for a specific class of Internet applications, viz. applications for distributed consensus. There

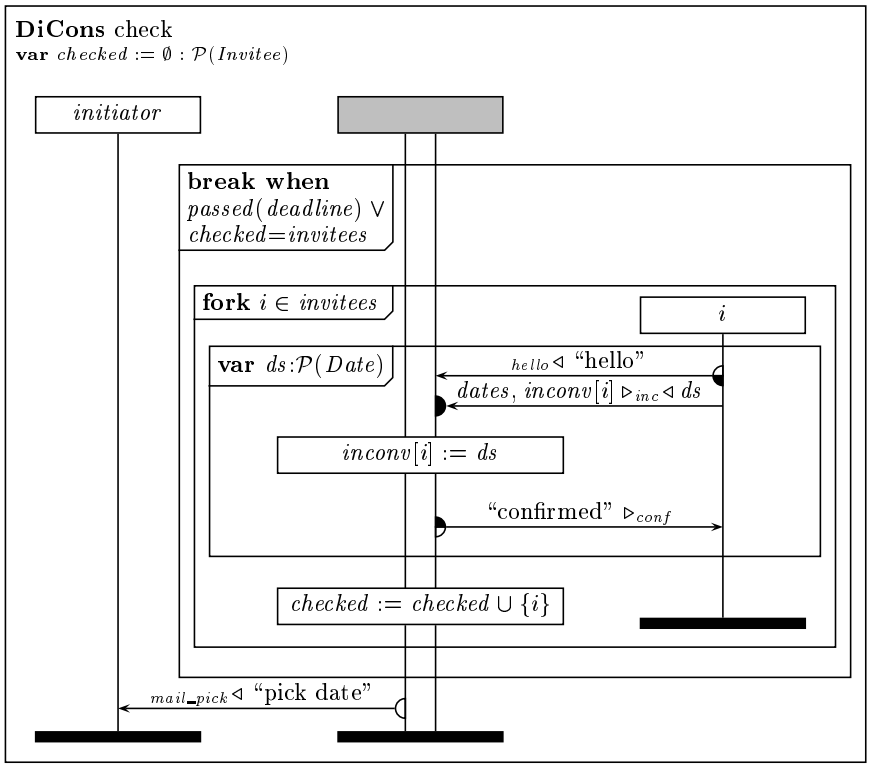

Fig. 17. Checking phase

are many different languages to specify Internet applications, but as far as we know, none of them is specifically designed to develop such applications.

Closest to our work is the development of the Web-language *Mawl* [1, 13]. This is also a language that supports interaction between an application and a single user. Mawl provides the control flow of a single session, but does not provide control flow across several sessions. 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.

Groupware [22] 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. On the other hand, asynchronous groupware is based on sending messages which do not have to be read and replied to immediately. An example of asynchronous groupware that can be specified in *DiCons* is a calendar for scheduling a project.

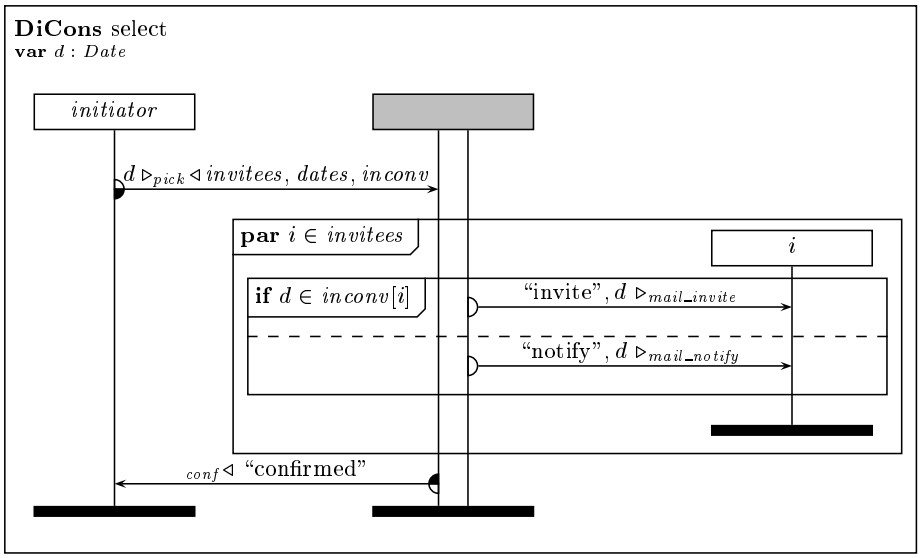

Fig. 18. Selection phase

Visual Obliq [4] 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*, an Obliq application can be distributed over several so-called sites on a number of servers.

Collaborative Objects Coordination Architecture (COCA) [14] is a generic framework for developing collaborative systems. In COCA, participants are divided into different roles, having different rights as in *DiCons*. Li, Wang and Muntz [15] used this tool to build an online auction.

Further, there are languages that allow programming of browsing behaviour. These allow, for instance, the behaviour of a user who wants to download a file from one of several mirror sites to be programmed. For so-called *Service Combinators* see [6, 12]. A further development is the so-called *ShopBot*, see [7].

However, none of the languages described above have a graphical representation.

5 Conclusions

We find that the nature of *DiCons* allows a complete MSC-like representation to be made in a straightforward manner. The examples show that although the specification does not become more compact, it does become clearer which agents communicate, and when they do so. This is exactly the strength of MSC.

In order to get a complete MSC representation, it was necessary to extend MSC with a number of new constructions. Maybe the presentation of these features is still a bit ad hoc, but the intention of this document is not to define

a graphical layer of *DiCons*, but rather to investigate the feasibility of such an exercise.

In *DiCons*, we group separate communication actions into one aggregated interaction. In the semantics, this leads to questions concerning atomicity. In general, research is needed to consider atomicity with respect to possible race conditions. Our aggregated interactions are a special case of the notion of message refinement as discussed in [8].

Using MSC gives a whole range of possibilities in terms of precision and completeness, in our setting. On the one hand, there are trace-based descriptions of runs, requirements or test traces of a system, on the other hand, there is the complete description of the whole behaviour of the server. Further, MSC supports all levels in between, so that it is also possible to describe the composition of test runs, or test purposes. The testing of Internet applications is still not developed very much, and is usually based on the manual execution of ad hoc test sequences. Now that we have a formal notation for a class of Internet applications, that both supports the description of complete systems and of the behaviour of components, it might be possible to start testing in a more formal and structured manner, for instance using Autolink [20] or TorX [21]. In this respect, it is interesting to see what is the relation between our graphical language and the MSC-based graphical language for TTCN-3, see [19].

We have implemented a compiler which can be used to compile (textual) *DiCons* specifications into Java Servlets [18]. See our Web site[1] for some working examples. Except for generating a Servlet, the compiler checks a specification on its syntax and static semantics. As making graphical tools for *DiCons* is quite a big investment, we will not do a complete implementation before the language *DiCons* has become stable.

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. J. Baeten, H. van Beek, and S. Mauw. Specifying internet applications with *DiCons*. In *Proceedings of the 16th ACM Symposium on Applied Computing (SAC 2001)*, Mar. 2001.
3. H. v. Beek. Internet protocols for distributed consensus – the *DiCons* language. Master's thesis, Eindhoven University of Technology, Aug. 2000.
4. 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.
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.

[1] The *DiCons* Web site can be found at *http://dicons.eesi.tue.nl/*.

7. 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.
8. A. Engels. Message refinement: Describing multi-level protocols in MSC. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC*, number 104 in Informatik-Berichte, pages 67–74, Berlin, Germany, June 1998. Humboldt-Universität zu Berlin.
9. A. Engels. Design decisions on data and guards in MSC2000. In S. Graf, C. Jard, and Y. Lahav, editors, *SAM2000. 2nd Workshop on SDL and MSC*, pages 33–46, Col de Porte, Grenoble, June 2000.
10. A. Engels, L. Feijs, and S. Mauw. MSC and data: Dynamic variables. In R. Dsoulli, G. von Bochmann, and Y. Lahav, editors, *SDL'99: The Next Millennium, Proceedings of the 9th SDL Forum*, pages 105–120, Montreal, Canada, June 1999. Elsevier.
11. ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC2000)*. ITU-TS, Geneva, 2000.
12. T. Kistler and H. Marais. WebL — a programming language for the Web. *Computer Networks and ISDN Systems*, 30(1–7):259–270, Apr. 1998.
13. 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.
14. 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.
15. D. Li, Z. Wang, and R. R. Muntz. Building web auctions from the perspective of collaboration. Technical report, UCLA Department of Computer Science, Sept. 1998.
16. S. Mauw and M. Reniers. High-level Message Sequence Charts. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 291–306, Evry, France, September 1997.
17. 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.
18. K. Moss. *Java Servlets*. Computing McGraw-Hill, July 1998.
19. E. Rudolph, I. Schieferdecker, and J. Grabowski. HyperMSC – a graphical representation of TTCN. In *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM'2000)*, Grenoble (France), June 2000.
20. M. Schimitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink – puting SDL–based test generation into practice. In A. Petrenko, editor, *Proceedings of the 11th International Workshop on Testing Comunicating Systems (IWTCS'98)*, pages 227–243. Kluwer Academic, 1998.
21. J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99:* 7^{th} *European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, Nov. 1999. EuroStar Conferences, Galway, Ireland.
22. J. Udell. *Practical Internet Groupware*. O'Reilly & Associates, Inc., Oct. 1999.