Operational semantics of *DiCons*, a formal language for developing Internet applications

J.C.M. Baeten H.M.A. van Beek 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

*DiCons* is a dedicated language designed to be able to develop Internet applications more efficiently and with better quality. We focus on applications in the area of distributed consensus: several users strive to reach a common goal. Formal methods help to improve quality, and allow proving properties and correctness. In this paper we develop a Plotkin-style operational semantics.

1 Introduction

Building Internet applications is not an easy task. Given the many problems involved, like session management and multi-user interaction, 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 [6] of which an extended abstract appeared as [4].

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 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.

The language *DiCons* needs a formal syntax so that we can use or build tools. Moreover, in order to avoid unclarities, ambiguities or misunderstandings it is vital that also a formal semantics is developed. This will make it possible to establish formal properties of *DiCons* applications. For instance, in the case of Internet voting, we want to prove that each participant only gets to vote once, and that the candidate with the most votes is elected.

In this article, we specify the formal syntax of *DiCons*, and provide a formal semantics using Plotkin-style operational rules. We also give a small example, and discuss future work.

2 *DiCons* Fundamentals

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*. First of all, we will discuss the restrictions
we make to the applications we want to specify. After that we show at what level of abstraction we would like to specify the applications, and we end this section by giving the interaction primitives we make use of in our specifications.

2.1 Restrictions

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 behavioural 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 behavioural expressions.

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 Level of Abstraction

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 Figure 1. In this drawing we sketch in MSC [16] a typical scenario of an Internet application which is called the Meeting Scheduler (see [21]). 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: An MSC scenario of an Internet application
```

The example shows that we have a central application, the server, and two roles, 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 side. The way in which a server can actively inform a user is e.g. by sending an e-mail. This interaction only contains information transmitted from the server to the user. The messages **options** and **convocation** can also be implemented as e-mails. 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. Sending information to the central application takes place in an interactive process, a so-called **session**. Messages **info** and **choice** can also best be implemented as sessions.

### 2.3 The Interaction Primitives

In **DiCons** we make use of interaction primitives which are at the level of abstraction explained in the previous section. First of all, the sending of messages like e-mails are constructed using **active push** primitives. This primitive is called an active push since from the viewpoint of the server, it actively pushes data to a user. We use a left-arrow (←) to textually specify this interaction primitive. Note that in the specifications, the user is placed on the left-hand side of the arrow, so the arrow is pointing to the user (push).

Since we use standard Internet technology, i.e. e-mail and Web browsers, we are committed to the HyperText Transfer Protocol (HTTP) [14] for most of the interactions between users and the central application. HTTP is based on a request-response way of interaction: a user has to request information from the server, e.g. by filling in a URL in a Web browser. The server then responds by sending a Web page or a Web form.

If the response on a user’s request contains a non-interactive Web page, the interaction is a standard page request which normally takes place when surfing on the Internet. We will call this kind of interactions **reactive pushes**. It is reactive since the server reacts on a user’s request and it is a push since the server pushes data to the user. For this primitive we make use of the right-left-arrow (⇛). This arrow shows that the user (which again is on the left-hand side) initiates the interaction after which the server (on the right-hand side) responds.

On the other hand, if the response to the user’s request contains a Web form, this form can be filled in and submitted by the user. This submission itself is another request, extended with some parameters containing the data filled in in the form. The server has to send a response on this new request, possibly containing another Web form. In this way, we can get a sequence of successive form submissions. This is called a **session**. Finally, after submission of the last form (which again is a request), the server responds by sending a Web page not containing a Web form, which ends the session as no submission can take place anymore. So a **session** is a sequence of form submissions, which is started by a user accessing the server via a URL request and ended by the server sending a plain Web page. In order to be able to clearly specify Internet applications from the viewpoint of the server, we want to couple the sending of a form to its submission. This leads to splitting up the sequence of HTTP request/response primitives, resulting in three kinds of **DiCons** interaction primitives. In Figure 2 an overview of the HTTP request/response sequence together with the **DiCons** interaction primitives is given.
The first interaction, a reactive pull, starts a session. This primitive specifies the URL request followed by the sending and submission of the first Web form. This interaction is represented using the right-left-right-arrow (→). The user starts the interaction, so it is called reactive. Apart from the data sent to the user there is also a flow of data from the user to the server. Therefore we call this interaction a pull: the system pulls data from the user via a Web form.

Subsequently, zero or more mid-session interactions can take place, which specify the sending and submission of subsequent forms. These interactions are called session-oriented pulls and are represented by a session-left-right arrow (→). The dashed part represents the preceding request (i.e. the preceding form submission). Again, this is called a pull since there is a flow of data from the user to the server. We call this interaction session-oriented because it is a session-based reaction on the submission of an earlier sent Web form.

Finally, for the ending of a session we make use of the session-oriented push primitive which specifies the sending of the final Web page. For this primitive we use the session-left-arrow (←) in which the dashed part again represents the preceding form submission. Again, we call this interaction session-oriented. It is called a push since the Web page sent to the user does not contain a Web form.

Looking at the five arrows one can easily see which arrows can be combined into sessions and which cannot. An arrow having a dashed tail can be connected to an arrow’s head pointing in the tail’s direction. In Figure 3 the way of combining interaction primitives into sessions is shown. Note that all correct interactions and sessions of interactions start with a solid tail and end with an arrow pointing to the left (i.e. to the user).

To summarize, we will give a short overview of the interaction primitives in Table 1.

### 3 The DiCons Syntax

In this section we will specify the syntax of DiCons using a BNF-like grammar. Since we focus in this paper mainly on the behavioural aspects of DiCons, we will present only the behavioural part of the language in detail.

We make some assumptions on the existence of several predefined or separately defined notions. These notions are needed to fully understand the grammar given below.
An active server push
An active push takes place if the server sends a message to a user (arrow to the left) which is not directly the result of a request from that user. So the server initiates the interaction which is indicated by the arrow’s tail pointing to the right.

A reactive server push
A reactive push takes place if the server sends a Web page (arrow to the left), not containing a Web form, to a user which is the result of a normal request from that user (user initiates, so tail to the left), i.e. not generated by filling out a previously received Web form.

A reactive server pull (a session start)
This interaction takes place if a user sends a request to the server (upper horizontal line, tail to the left) on which the server responds by sending a Web form (middle line). This form is filled in and submitted by the user (lower line, arrow to the right). A reactive pull starts a session with one particular user.

A reactive server pull (within a session)
In response to a prior form submission (upper, dashed line), the server sends a Web form to the user (middle line). Subsequently, the user submits the filled in form (lower line, arrow to the right). This interaction is repeated in the middle of a session.

A session-oriented server push (session end)
The server sends a non-interactive Web page to the user (lower line) in response to a prior form submission by the user (upper, dashed line). This interaction ends a session since no Web form can be filled in anymore.

Table 1: The DiCons interaction primitives

First of all, we have several assumptions on the existence of data types. At least Booleans, but possibly also other standard types, such as Naturals, (with appropriate constants and operations) are supposed to be available. We also assume mechanisms to compose data types into more complex data types (see the non-terminal `type expression`). For each of these data types we have an interpretation in an appropriate semantical domain.

There is one special data type, which is the universe of all potential interaction partners (i.e. the class of all users that can possibly communicate with the system). We denote this universe by $U$. Subsets of this class are called roles. So a role is a set containing users. We assume that the standard set operations are defined on roles.

Next, we assume the existence of a collection of typed variables. Using variables, constants and operations, we can make typed expressions.

In addition to assignments to single variables, we frequently encountered the need for procedural subprograms in our examples. Therefore, we also assume that a number of (typed) procedures is given, which are specified by the effect which the procedure has on its parameters. In Table 4 (rule 4) we denote this function by $\text{eff}$. Procedures are executed as atomic actions and only affect (values of) their parameters. From the viewpoint of simplicity we restrict ourselves to only use procedures with their input and output parameters separated using a semicolon.

In order to keep the grammar given below simple, we do not specify the exact interpretation of an (identifier), nor do we give nice syntax to reduce the abundance of parentheses in the definition of a (process). We simply follow the normal convention that parentheses may be omitted whenever possible without introducing ambiguities. Therefore, we assume that, after the monadic process operators, the sequential composition operator binds strongest and that the other operators all have the same priority.

Finally, we presume the existence of a data type called Time. This simply is a discrete, totally ordered set. The (dynamic) constant now is used to refer to the current time. By means of relational expressions over this constant and regular elements of the time domain, we can form
The conditional branching operator is denoted with the condition in the middle. The next two operators are the conditional repetition operator and the conditional disrupt operator. The conditional repetition behaves like the well-known while loop construct from regular programming languages. The conditional disrupt will stop the execution of the included process expression at the moment that the Boolean expression becomes true. Next, we have three parallel operators. The first two are simply the interleaving operator and its generalization over collections of users. The third operator is the generalized replication operator. For every user in its domain, any number of instances of the given process can be forked in parallel.

We consider the five types of interactions as explained in Section 2.3.

### Table 2: BNF specification of the DiCons syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>BNF Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{diconsapp} )</td>
<td>( \text{::= } { \text{{'roledecl} '</td>
</tr>
<tr>
<td>( \text{roledecl} )</td>
<td>( \text{::= } [\text{{rolename} {, \text{{rolename}}}*]} )</td>
</tr>
<tr>
<td>( \text{rolename} )</td>
<td>( \text{::= } {\text{{identifier}}} )</td>
</tr>
</tbody>
</table>
| \( \text{process} \) | \( \text{::= } \{\text{( \{vardecl\} | \{process\} \}{, \{process\} \)} \} | \{\text{( \{boolean expression\} \{\text{\{\{'domaindecl\} \{process\}'\} \)+} \text{\{process\}'\} \} | \{\text{( \{boolean expression\} \{\text{\{domaindecl\} \{process\}'\} \} \} | \{\text{( \{process\} \{\text{\{domaindecl\} \{process\}'\} \} | \{\text{\{domaindecl\} \{process\}'\} \} | \{\text{\{interaction\} \{\text{\{local action\} \{\text{\{\{'varlist\}'\} \}} \} | \{'\varepsilon' \}
|
| \( \text{vardecl} \) | \( \text{::= } \{\text{\{varname\} \{\rightarrow \text{\{expression\}}\} \}{, \{\text{\{type expression\}}\} \} \} \) |
| \( \text{varname} \) | \( \text{::= } \{\text{\{identifier\}}\} \) |
| \( \text{domaindecl} \) | \( \text{::= } \{\text{\{varname\} \{\epsilon\}' \{\{U\} | \{\text{\{rolename\}}\}\} \} \) |
| \( \text{interaction} \) | \( \text{::= } \{\text{\{\{'bind\}' \{\text{\{varname\}}\} \{, | \{\rightarrow \text{\{\{interaction\}} \} \} | \{'\varepsilon' \}
|
| \( \text{interactname} \) | \( \text{::= } \{\text{\{identifier\}}\} \) |
| \( \text{inputparams} \) | \( \text{::= } \{\text{\{varlist\}}\} \) |
| \( \text{varlist} \) | \( \text{::= } \{\text{\{varname\} \{, \text{\{varname\}}\}*]\} \) |
| \( \text{uservarname} \) | \( \text{::= } \{\text{\{identifier\}}\} \) |
| \( \text{outputparams} \) | \( \text{::= } \{\text{\{expressionlist\}}\} \) |
| \( \text{expressionlist} \) | \( \text{::= } \{\text{\{expression\} \{, \text{\{expression\}}\}*]\} \) |
| \( \text{local action} \) | \( \text{::= } \{\text{\{'action\} \{'\{\text{\{action\}}\}'\} \} \) |
| \( \text{action} \) | \( \text{::= } \{\text{\{varname\} \{\rightarrow \text{\{expression\}}\} \} | \{\text{\{proccname\} \{\{'\{\text{\{inputparams\}} \{, | \{\text{\{outputparams\}}\}'\} \} \) \} \) |
| \( \text{proccname} \) | \( \text{::= } \{\text{\{identifier\}}\} \) |
A local action is either an (atomic) assignment to a variable, or the (atomic) execution of a procedure. The naming of the parameters of procedures is as seen from the viewpoint of the server. So an input parameter is a value-result parameter and an output parameter a value parameter. In order for syntactically correct DiCons programs to make sense, a number of additional requirements must be met. We assume that all variables are declared before use and that all expressions can be typed correctly. Furthermore, a variable used for specifying a user within an interaction \((\text{uservarname})\) must be of type \(U\). The order in which session-related interactions take place must answer their semantics as explained in Section 2.3. Finally, we restrict the occurrence of parameters in several interactions in the following way: A server push (i.e. one of \(\leftarrow\), \(\Rightarrow\), and \(\varpi\), see Section 2.3) is not allowed to have input parameters.

4 Operational Semantics of DiCons

In this section we give the operational semantics of DiCons using Structured Operational Semantics rules, firstly introduced by Plotkin [25]. We define the operational semantics on the BNF given in Section 3. We use auxiliary actions to decompose the interaction primitives (see Section 4.2).

In our specification we make use of the naming scheme for the identifiers given in Table 3. We use the vector notation to specify a list of identifiers. Apart from vectors we can also use primes and subscript notations to introduce new identifiers.

Since we make use of a scope operator which can introduce local scopes within an application we must keep track of the variables that are available for each (sub)process. Since scopes can be nested, it can occur that several variables with the same name are declared. To solve the problem of choosing the right variable, e.g. for an assignment, we use the technique of extending each process with a state stack as specified in [9]. This technique is widely used in traditional compiler implementations. In this state stack we keep track of the variables declared in the scope of the process and the order in which these declarations took place.

4.1 Variable declarations and stacks

In DiCons we can declare (local) variables using the scope operator \([s \mid p]\). The declaration \(s\) is of the form \(x \mapsto e : T\) or \(x : T\). If declaration \(s = x \mapsto e : T\) takes place in a state \(\sigma\), \(e\) is evaluated in \(\sigma\) and its result, the valuation \(x \mapsto \sigma(e) : T\), is added to the state space. A declaration \(s = x : T\) is added to the state space without evaluation. During the execution of \(p\) the mapping of variable \(x\) to its value can change. If process \(p\) comes to an end, the valuation is popped from the stack. We use \(\lambda\) to denote the empty state stack.
Definition 1 (Valuation) Let \( x \) be a variable, \( c \) be a constant and \( T \) be a type. A valuation \( s \) is then defined by

\[
\begin{align*}
s & ::= x \mapsto c : T \quad \text{defined valuation} \\
& | \quad x : T \quad \text{undefined valuation}
\end{align*}
\]

Definition 2 (State stack) Let \( s \) be a valuation. A state stack \( \sigma \) is then defined by

\[
\begin{align*}
\sigma & ::= \lambda \quad \text{the empty state stack} \\
& | \quad s :: \sigma \quad \text{a nonempty state stack}
\end{align*}
\]

Definition 3 (Evaluation) Let \( \sigma \) be a state stack, \( s \) be a valuation, and \( x \) be a variable. The evaluation function \( (s :: \sigma)(x) \) is a partial function. It is not defined if \( x \) does not occur in \( (s :: \sigma) \) or if the binding occurrence of \( x \) is uninitialized. Therefore, we can define evaluation \( (s :: \sigma)(x) \) by the following equation:

\[
(s :: \sigma)(x) = \begin{cases} 
  c & \text{if } \exists \exists T \ s = (x \mapsto c : T) \\
  \sigma(x) & \text{if } \neg \exists \exists T \ s = (x \mapsto c : T) \lor s = (x : T)
\end{cases}
\]

We extend the evaluation operator to lists of variables \( \sigma(y) \).

Furthermore, we want to be able to substitute valuations in a state stack. A substitution can change a variable’s mapping with a mapping of the same variable to another value. Substitution \( \sigma[c/x] \) replaces the uppermost occurrence of a valuation of variable \( x \) in state stack \( \sigma \) (i.e. the occurrence with the smallest scope) with a new valuation where \( x \) is mapped to value \( c \).

Definition 4 (Substitution) Let \( \sigma \) be a state stack, \( s \) be a valuation, \( T \) be a type, \( x \) be a variable of type \( T \), and \( c \) and \( c' \) be constants in \( T \). A substitution is then defined by

\[
\begin{align*}
\lambda[c/x] & = \lambda \\
(s :: \sigma)[c/x] & = \begin{cases} 
  (x \mapsto c : T) :: \sigma & \text{if } s = (x \mapsto c' : T) \lor s = (x : T) \\
  s :: (\sigma[c/x]) & \text{otherwise}
\end{cases}
\end{align*}
\]

Likewise, we define substitution of lists of variables \( \sigma[e/y] \).

4.2 Actions

In the DiCons language we have two different sets of actions. First of all, we have the actions which we use to specify DiCons applications. These actions can contain variables which must be declared using the state operator (Section 4.1) or in the domain part of of an interleaving or replication operation (Section 4.3.6). A description of the choice for the arrows is given in Section 2.3. The actions are given below:

1. \text{action}(x := e) \quad \text{The assignment of the value of expression } e \text{ to variable } x.

2. \text{action}(P(\bar{e}; \bar{c})) \quad \text{The execution of procedure } P \text{ with input (value-result) parameters } \bar{e} \text{ and output (value) parameters } \bar{c}.

3. \text{u} \leftarrow m(\bar{c}) \quad \text{An active push: the sending of an e-mail } m \text{ with parameters } \bar{c} \text{ to user } u.

4. \text{u} \Rightarrow m(\bar{c}) \quad \text{A reactive push: the sending of a Web page } m \text{ with parameters } \bar{c} \text{ to user } u.

5. \text{bind } x \Rightarrow m(\bar{c}) \quad \text{A reactive push: the sending of a Web page } m \text{ with parameters } \bar{c} \text{ to a user that is stored in } x.
6. \( u \Leftarrow m(\vec{y}; \vec{c}) \) A reactive pull: the sending and submission of a Web form \( m \) with input parameters \( \vec{y} \) and output parameters \( \vec{c} \) to user \( u \) as response to a URL request.

7. \( \text{bind } x \Leftarrow m(\vec{y}; \vec{c}) \) A reactive pull: the sending and submission of a Web form \( m \) with input parameters \( \vec{y} \) and output parameters \( \vec{c} \) to a user that is stored in \( x \) as response to a URL request.

8. \( u \Leftarrow m(\vec{y}; \vec{c}) \) A session-oriented pull: the sending and submission of a Web form \( m \) with input parameters \( \vec{y} \) and output parameters \( \vec{c} \) to user \( u \) as response to a prior submission of a Web form.

9. \( u \Leftarrow m(\vec{c}) \) A session-oriented push: the sending of a Web page \( m \) with output parameters \( \vec{c} \) to user \( u \) as response to a prior submission of a Web form.

Actions 1 to 3 are atomic actions, whereas actions 4 to 9 have substructure. This substructure is defined in terms of the atomic actions 10 and 11. These two actions can be seen as auxiliary actions.

10. \( u \Rightarrow \text{req}(\vec{y}) \) A URL request or form submission by user \( u \) with input parameters \( \vec{y} \). Actually, this represents an HTTP request to receive a web page.

11. \( u \Leftarrow m(\vec{c}) \) The sending of a Web page \( m \) with output parameters \( \vec{c} \) to user \( u \).

We proceed to give operational rules. The states consist of three fields: first, we have a DiCons expression. Second, we have a state stack. This keeps track of values of variables, respecting the scope of local variables. Third, we have a time component. Time is measured on a discrete scale. In the transition system, time can always proceed without changing the state of a system. Process behaviour is not affected directly by passage of time, thus each expression allows passage of time without change. However, boolean expression can contain the (dynamic) now constant and therefore evaluate different due to passage of time. By use of the conditional operators, this in turn affects process behaviour.

Next, the transitions have two labels. Above the arrow we have the action labels. These are taken from the following set:

\[
\begin{align*}
\text{assign}(x, c) & \quad \text{exec}(P(\vec{x}, \vec{d}, \vec{c})) & \quad \text{mail}_{x}(m(\vec{d})) & \quad \to_{x}(m(\vec{d})) & \quad \text{req}_{x}(\vec{x}, \vec{d})
\end{align*}
\]

If vectors are empty, they can be omitted. By looking at rules 3, 4, 5, 12 and 13 in Table 4 one can easily see the one-on-one mapping of atomic actions to transition labels.

Note that all expressions in these labels except variables \( x \) and \( \vec{x} \) are evaluated to constant values (also, we keep track of variables that receive new values). Below the arrow we keep track of parallel processes. It is possible that a user has several sessions running in parallel, using the interleaving (\(||\)) or the replication (\(!\)) operator. The interactions taking part in these interleaved sessions must be labeled in such a way that the corresponding sessions can be uniquely determined.

In order to keep the actions from different sessions separated, we use a sequence of bits. Each bit denotes either a left-hand or a right-hand side in a parallel composition.

We make use of the down-arrow (\(\downarrow\)) to indicate termination [5]. The Structured Operational Semantics of the atomic actions are given in Table 4.
Function $\sigma_t$ returns the evaluation of its parameter in state $\sigma$ at time $t$. If such an evaluation $\sigma_t(e)$ is not defined then that is caused by the occurrence of a variable in $e$ which has no value in $\sigma_t$. This means that the SOS rule in which this evaluation is assumed cannot fire. In that case we have a live-lock situation which models a run-time error. Furthermore, function $type_{\sigma}$ returns the type of its parameter in state $\sigma$.

As an example we will explain SOS rule 8 in more detail. As described in item 6 on page 9, a reactive pull consists of three parts, viz. a URL request, followed by the sending of a web form (containing the evaluated output parameters of the interaction) and the submission of this form with the input parameters filled out. The first action is given as a transition label above the transition arrow. The other two actions are given as atomic actions in the process expression. These atomic actions themselves give rise to transitions using SOS rules 12 and 13.
4.3 Operators

4.3.1 State operator

We make use of the state operator \([s \mid p]\) to declare variables with a local scope. A variable declared in \(s\) is only available to process \(p\). The SOS rules for the state operator are given in Table 5.

\[
\begin{align*}
\text{Table 5: SOS rules for the state operator}
\end{align*}
\]

4.3.2 Sequential composition

We make use of the basic operator for sequential composition \((\_ \cdot \_)_\) as defined in [5]. The SOS rules for these operators are given in Table 6.

\[
\begin{align*}
\text{Table 6: SOS rules for sequential composition}
\end{align*}
\]

4.3.3 Conditional branching

We use the conditional operator \((\_ \triangleright \_ \triangleright \_)_\), as defined in [15]. This operator behaves like the if-then-else-fi operator in sequential programming: \(p \triangleleft b \triangleright q \equiv \text{if } b \text{ then } p \text{ else } q \text{ fi.}\) The SOS rules for conditional branching are given in Table 7. Note that we make use of \(tt\) and \(ff\) to represent boolean values true and false.
4.3.4 Conditional repetition

The conditional repetition can be compared to a while loop in traditional programming. We make use of the conditional repetition operator \((b \text{ while } p \text{ do } \text{ od})\) to specify these repetitions: \( b \text{ while } p \equiv \text{ while } b \text{ do } p \text{ od} \). The while loop repeats statement \( p \) until test \( b \) proves false. The SOS rules for the conditional repetition are given in Table 8.

| \( \langle p, \sigma, t \rangle \downarrow, \quad \sigma_i(b) = \text{tt} \) | \( \langle p, \sigma, t \rangle \overset{a}{\kappa}(p', \sigma', t), \quad \sigma_i(b) = \text{tt} \) |
| \( \langle p \triangleleft b \triangleright q, \sigma, t \rangle \downarrow \) | \( \langle p \triangleleft b \triangleright q, \sigma, t \rangle \overset{a}{\kappa}(p', \sigma', t) \) |
| \( \langle q, \sigma, t \rangle \downarrow, \quad \sigma_i(b) = \text{ff} \) | \( \langle q, \sigma, t \rangle \overset{a}{\kappa}(p', \sigma', t), \quad \sigma_i(b) = \text{ff} \) |
| \( \langle p \triangleleft b \triangleright q, \sigma, t \rangle \downarrow \) | \( \langle p \triangleleft b \triangleright q, \sigma, t \rangle \overset{a}{\kappa}(p', \sigma', t) \) |

Table 7: SOS rules for conditional branching

4.3.5 Conditional disrupt operator

We introduce the \( b \triangleright p \) statement to specify conditional disrupts. This means that process \( p \) is normally executed until \( b \) becomes true. At that moment the statement terminates, independent of the (inter)actions that are taking place at that moment. If \( p \) terminates before \( b \) becomes true the statement terminates 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. The SOS rules for the conditional disrupt operator are given in Table 9.

| \( \langle p, \sigma, t \rangle \overset{a}{\kappa}(p', \sigma', t), \quad \sigma_i(b) = \text{tt} \) | \( \sigma_i(b) = \text{ff} \) |
| \( \langle p \triangleright p, \sigma, t \rangle \overset{a}{\kappa}(p', (b \triangleright p), \sigma', t) \) | \( \langle b \triangleright p, \sigma, t \rangle \downarrow \) |

Table 8: SOS rules for conditional repetition

4.3.6 Parallel composition

We have the merge operator || denoting parallel composition. We use the notation with three lines to emphasize the fact that parallel components do not interact. Each session is independent, only communicates with the central server. In order to keep parallel sessions with the same user separate, it is necessary to keep track of the origin of an action in a parallel composition. We use a sequence of bits for this, with a 0 denoting the left component, and a 1 the right component. In essence, this is the same labeling as used by [12] and others, often called locations.

In addition, we have two operators for parallel composition at our disposal: the replication or bang operator (!) and the interleaving or merge operator (||). Process \( ! \in \text{Rep}(u) \) means that
all users $u \ (u \in R)$ can execute (inter)actions in process $p(u)$ between user $u$ and the central applications in parallel and more than once. On the other hand, process $\|_{u \in R} p(u)$ specifies that all users $u \ (u \in R)$ will execute (inter)actions in process $p(u)$ between user $u$ and the central applications in parallel but only once. So, in contrast to the replication operation, after execution of $p(u)$ for all $u \in R$ the interleaving operation terminates. Moreover, if $p(u)$ terminates for all $u \in R$ or if the set of users $R$ is empty, then both $\|_{u \in R} p(u)$ and $\|_{u \in R} p(u)$ terminate.

The SOS rules for these operators are given in Table 10.

Table 10: SOS rules for the interleaving composition

\[
\begin{align*}
\langle p, \sigma, t \rangle & \downarrow, \quad \langle q, \sigma, t \rangle \downarrow & \quad \langle p, \sigma, t \rangle^{\alpha_k} \langle q', \sigma', t \rangle^{30} \\
\langle p \| q, \sigma, t \rangle & \downarrow & \quad \langle p \| q, \sigma, t \rangle^{\alpha_{k_0}} \langle q', \sigma', t \rangle^{\alpha_{k_1}} \langle q \| q', \sigma', t \rangle^{31} \\
\forall c \in R \langle p(c), \sigma, t \rangle & \downarrow & \quad \langle p \| p(x), \sigma, t \rangle \downarrow^{32} \\
\langle p \| p(x), \sigma, t \rangle & \downarrow & \quad \langle p \| p(x), \sigma, t \rangle^{\alpha_{k_0}} \langle p', \sigma', t \rangle^{\alpha_{k_1}} \\
\forall c \in R \langle p(c), \sigma, t \rangle & \downarrow & \quad \langle p(x), \sigma, t \rangle \downarrow^{34} \\
\langle p(x), \sigma, t \rangle & \downarrow & \quad \langle p(x), \sigma, t \rangle^{\alpha_{k_0}} \langle p', \sigma', t \rangle^{\alpha_{k_1}} \\
\forall c \in R \langle p(c), \sigma, t \rangle & \downarrow & \quad \langle p(x), \sigma, t \rangle \downarrow^{35} \\
\langle p(x), \sigma, t \rangle & \downarrow & \quad \langle p(x), \sigma, t \rangle^{\alpha_{k_0}} \langle p', \sigma', t \rangle^{\alpha_{k_1}}
\end{align*}
\]

5 Why an operational semantics?

In the design of a domain-specific language, such as DiCons, the development of syntax, semantics and pragmatics goes hand in hand (see [22]). In order to conduct experiments, a syntax is needed. In order to explore the problem domain, semantical analysis is needed and in order to understand the practical applicability, pragmactical case studies must be performed. These phases influence each other. The current stage of DiCons is that a first syntax has been developed, together with a collection of examples and some tool support. However, the semantics of the language was only expressed in an informal way. In the current paper we have brought the development of the DiCons language one step further by providing it with an unambiguous semantics. We list some virtues of our semantical work.

Unambiguous operational meaning The major result of our work is that DiCons expressions now have an unambiguous operational meaning. This is a prerequisite for building tools and analyzing example specifications. We are unaware of any other formal treatment of the communication primitives that underly the client-server approach on the Internet.

Property definition and analysis A formal semantics allows us to define and analyse specific properties of Internet applications. One such property, discussed in Section 7.3, is the notion of independent responsiveness: the time before which an application responds to a user’s request does not depend on interactions that take place between the application and other users. Capturing such notions is essential to building proper applications.

Tool specification A formal definition of a language can be used as a specification or reference implementation of support tools. We are currently working on updating the compiler and make it comply exactly with the semantics given in this paper. A simulator, based on the operational semantics, helps to explore the operational behaviour of an application. Such a tool could e.g. be linked to a tool for automatic test generation and in this way support the testing of applications which were built independently of DiCons (cf. the TorX tools [7]).
\begin{center}
\begin{verbatim}
(Candidates, Voters | 
[i: U; election-name: String; nomination-deadline, voting-deadline: Time; | 

-- initialization 
bind +start-election(); 
i +election-data(election-name, Voters, nomination-deadline, voting-deadline;); 
(\forall v \in Voters \; v \leftarrow invite-nomination(election-name, nomination-deadline);) 
i \leftarrow voting-initialized(); 

-- nomination 
(now > nomination-deadline \Rightarrow 
(\forall v \in Voters \; v \leftarrow my-nomination(); 
  \textbf{action}(Candidates := Candidates \cup \{v\}); 
  v \leftarrow nomination-accepted();)) 

(results: array Candidates of Integer | 

-- voting 
\textbf{action}(initialize(results)); 
(\forall v \in Voters \; v \leftarrow invite-vote(election-name, voting-deadline, Candidates);) 
(now > voting-deadline \Rightarrow 
(\forall v \in Voters \; C. Candidates \mid v \leftarrow vote(c, Candidates); 
  \textbf{action}(add-vote(results,c)); 
  v \leftarrow vote-accepted();)) 

-- results 
\forall C. Candidates | 
  \textbf{action}(select-winner(c,results)); 
  (\forall v \in Voters \; v \leftarrow congratulations(); \& c=v \Rightarrow v \leftarrow winner-is(c)); 
i \leftarrow voting-finished(c, results) 
)
\

Example 1: voting via Internet
\end{verbatim}
\end{center}

One could even link model checking tools to the simulation tool, such that properties could be verified automatically.

6 Example: voting via Internet

In [4, 3] we have studied the design of a typical Internet application, called the Meeting Scheduler. It nicely shows the use of most of the language constructs in DiCons. This example can be easily adapted to follow the more mathematically oriented notation used in the current paper. Here, we will give a different example to demonstrate the use of DiCons: an Internet application to support voting (Example 1).

The purpose of the application is to support a group of people in electing one of them. We consider three kinds of users. The first user is the initiator who starts the election process. The second kind of users are collected in role Candidates, which contains all people who nominate themselves for the election. Finally, the role Voters contains the users invited by the initiator to take part in the election process. The process comprises four phases.

The first phase is the initialization phase in which the initiator requests to start an election. Since the initiator takes the initiative, this is modeled by a reactive server pull (\texttt{\&}). This is the
first interaction of a session with initiator $i$. The *bind* construct indicates that the identity of the user initializing this interaction will become bound to variable $i$. Next, the initiator is prompted to supply information about the election, such as the election name, the people allowed to take part, the deadline for nominating candidates, and the deadline for voting. The interaction is therefore a session oriented server pull ($\rightarrow$). Having received this information, the system can inform the voters that a voting will take place, using an active server push ($\leftarrow$). The session with $i$ is ended by a confirmation ($\rightarrow$).

The second phase concerns the collection of nominations. This phase will be interrupted when the nomination deadline passes. During this phase every voter is allowed to nominate himself, after which his identity is added to the set of candidates. Note that, due to the use of the replication operator, every voter can nominate himself more than once. This is not a problem because we use set union when extending the set of candidates.

In the third phase the actual voting takes place. First, we need to declare some data structure to collect the voting information. This can e.g. be implemented as an associative array, which for every candidate keeps track of the number of votes for that candidate. These results are set to zero through calling procedure *initialize*. After informing the voters that the voting phase has started, they have until the voting deadline to submit their votes. Every voter can vote at most once. The variable *results* is accordingly updated by calling the procedure *add-vote*.

In the last phase, the winner is selected by inspecting the collected votes (we assume that procedure *select-winner* will also select a unique winner in case of a draw). Finally, the results of the voting are distributed among all voters.

The *DiCons* specification of the voting process given in Example 1 is incomplete in several ways. The reason is that we focus in this paper on the behavioural part of the language. The complete *DiCons* language contains means to specify this additional information. We will summarize the information that should be provided to make this example fully operational.

First of all, we have only informally described the data types and procedures involved. We can use any current programming or data specification language for this purpose. In order to be able to easily experiment with *DiCons* we have included the Java programming language for this purpose.

Next, the interactions are stated in a very abstract way: just by providing the interaction partner, the type of interaction, its name and parameters. In *DiCons* these interactions are made operational by means of *presentations*. A presentation describes how the parameters are embedded in, e.g., an e-mail or a web form, based on the type and name of the interaction. The *DiCons* run-time system dynamically generates the corresponding e-mail or web pages.

Finally, we mention security aspects which are not taken into consideration in this example specification. The complete *DiCons* language supports an authentication mechanism based on user names and passwords. Further support for preserving confidentiality and integrity of data is planned but not yet operational.

### 7 Properties of Internet Applications

In this section we specify and verify some properties of Internet applications. Properties can be divided into two classes: general properties and application-dependent properties. General properties are properties of all Internet applications we focus on where application-dependent properties are subject to a specific application.

Before we give these properties we explain the notion of blocking, which is an important concept with respect to (multi-user) Internet applications in general. Apart from that, we make some assumptions on the users that make use of the application and the robustness of the server.

#### 7.1 Blocking

An interaction can be blocked by a user if it is waiting for that user to interact with it. I.e., if it is waiting for the user to send a new request. In Section 2.3 we explained the way in which
interaction primitives are constructed of HTTP requests and responses. From that point of view, an interaction can be blocked if it is waiting for the user to send a URL request or to fill in and submit a Web form. To be able to formally specify the blocking concept we first introduce the way we reason about traces. This is based on [2]. As explained in Section 4.2 we label our transitions using several action labels. Apart from the action labels we add a session identifier to the transitions to be able to keep the actions within different sessions separated. In our semantics, a trace is a sequence of these action label/session identifier combinations which correspond with actions that subsequently can take place starting in some initial state.

**Definition 5 (Trace)** Let $a$ be an action label and $k$ be a session identifier. A trace is $\alpha$ then defined by

$$
\alpha ::= \lambda \quad \text{the empty trace} \\
| \langle a, k \rangle; \alpha \quad \text{a nonempty trace}
$$

We have some general functions on traces for determining the head, tail and length of a trace. Note that both the head and tail functions are partial. Let $\alpha$ be a trace, $a$ be an action label and $k$ be a session identifier, then

$$
\begin{align*}
\text{head}(\langle a, k \rangle; \alpha) &= \langle a, k \rangle \\
\text{tail}(\langle a, k \rangle; \alpha) &= \alpha \\
|\alpha| &= \begin{cases} 
0 & \text{if } \alpha = \lambda \\
1 + |\text{tail}(\alpha)| & \text{otherwise}
\end{cases}
\end{align*}
$$

Apart from these general functions we have some specific functions in combination with our semantics. The first function we introduce is for determining whether a user is blocking a trace within a particular session. This means that the trace starts with a request label.

**Definition 6 (Blocking)** Let $\alpha$ be a trace, $c$ be a (constant) user and $k$ be a session identifier. The property of a trace being blocked by a user $c$ in session $k$ is then defined by

$$
\exists c, k \left( \text{head}(\alpha) = \langle \text{req}, \langle \tilde{y}, \tilde{d} \rangle, k \rangle \right)
$$

Given these functions we can only reason about predefined traces. However we want to reason about processes in a given state at a given time. Therefore we want to transform a process into a (possibly infinite) set of feasible traces. We do this using the $\text{tr}$ function which returns all traces that can take place starting a process in a given state at a given time. Traces can be infinitely long if the process does not terminate.

**Definition 7 (Traces of a process)** Let $p, p'$ be processes, $\sigma, \sigma'$ be state stacks, $t, t'$ be time stamps, $a$ be an action label and $k$ be a session identifier. At time $t$ in state $\sigma$, the set of traces of process $p$ is then defined by

$$
\text{tr}(p, \sigma, t) = \{ \langle a, k \rangle; \alpha \mid \langle p, \sigma, t \rangle \xrightarrow{a}_k \langle p', \sigma', t' \rangle \land \alpha \in \text{tr}(\langle p', \sigma', t' \rangle) \}
$$

Moreover, $\lambda \in \text{tr}(p, \sigma, t)$ if and only if $\langle p, \sigma, t \rangle \downarrow$.

A process is blocked by a user if in a given state a trace exists that is blocked by that user.

**Definition 8 (Blocking of a process)** Let $p$ be a process, $\sigma$ be a state stack, $t$ be a time stamp and $c$ be a constant user. At time $t$ in state $\sigma$, a process $p$ is being blocked by user $c$, is then defined by

$$
\exists \alpha \in \text{tr}(p, \sigma, t) \exists k \big( \exists c, k \big)
$$
Apart from the application being blocked, a user can be blocked by an application. This takes place between the sending of a request and the receiving of a response. So a user is blocked if he is waiting for a reaction to a session-oriented message from the server.

If a given specification answers the syntax and static assumptions, this means that a user is blocked by a process in a given state at a given time if there is a trace of that process in which the first occurrence of the receiving of a reactive or session-oriented message in a session precedes the first occurrence of the sending of a request in that session. To determine the number of steps before the first occurrence of a given transition takes place, we make use of the $\#_1$ operator.

$$
\#_1((a,k), \alpha) = \begin{cases} 
\infty & \text{if } \alpha = \lambda \\
1 & \text{if } \text{head}(\alpha) = (a,k) \\
1 + \#_1((a,k), \text{tail}(\alpha)) & \text{otherwise}
\end{cases}
$$

Again, we first specify a function over traces which we use for specifying the function over processes.

**Definition 9** (User-blocking for traces) Let $\alpha$ be a trace, $c$ be a (constant) user and $k$ be a session identifier. The property of a user $c$ being blocked in session $k$ by a trace $\alpha$ is then defined by

$$
\boxdot_{c,k} \equiv \exists \bar{d} \exists \bar{g} \exists \bar{d}' \left( \#_1((to_c(m(\bar{d})), k), \alpha) < \#_1((req_c(\bar{g}, \bar{d}'), k), \alpha) \right)
$$

**Definition 10** (User-blocking for processes) Let $p$ be a process, $\sigma$ be a state stack, $t$ be a time stamp and $c$ be a (constant) user. At time $t$ in state $\sigma$, the property of user $c$ being blocked by process $p$ is then defined by

$$
\boxdot(p, \sigma, t) \equiv \exists \alpha \in \text{tr}(\langle p, \sigma, t \rangle) \exists k \boxdot_{c,k}
$$

To be able to prove properties of processes, it would be very useful to introduce the notion of invariants for traces and therefore for processes. To be able to verify properties after execution of each action means that we want to be able to verify properties at all semicolons in a trace. This can be done by proving that the property holds at the beginning of all possible postfixes (tails) of a trace. By using the tails function we can generate this set of tails.

$$
tails(\alpha) = \begin{cases} 
\{ \alpha \} & \text{if } \alpha = \lambda \\
\{ \alpha \} \cup \text{tails}(\text{tail}(\alpha)) & \text{otherwise}
\end{cases}
$$

We will use this function for defining the properties in section 7.3.

### 7.2 Assumptions

Since we have a notion of blocking, we have to make some assumptions on interactions to be able to define suitable properties of Internet applications. Furthermore, we make an assumption on the robustness of the server.

**User activity** Users receive and read messages sent by an active push interaction.

**User interactivity** Web forms sent to a user are finally filled in and submitted. We need this assumption to prevent the application from being blocked.

**Server robustness** The server on which the application runs will not crash or shut down as long as the application is active.

Note that we only make assumptions with respect to the user and the robustness of the server. We do not make any assumptions on aspects of the application itself. Since we want to formally verify specifications of Internet applications we cannot and will not introduce these properties in our semantics. This would make it unnecessarily complex.
7.3 An example: independent responsiveness

In this section we will prove the independent responsiveness (IR) property to show the usefulness of DiCons. This property is a general property of Internet applications which must always hold.

Users who interact with the server can block the application as explained in Section 7.1. However, we do not want users to be able to block other user’s sessions.

**Definition 11 (Independent responsiveness)** Let \( p \) be a process, \( \sigma \) be a state stack, \( t \) be a time stamp. At time \( t \) in state \( \sigma \), the independent responsiveness IR property for process \( p \) is then defined by

\[
IR((p, \sigma, t)) = \forall \alpha \in \text{tails}(tr((p, \sigma, t))) \neg(\exists c \exists d \exists k (c \neq d \land [\text{DiCons}_c, k] \land [\text{DiCons}_d, k]))
\]

The IR property says that for all traces in the set of tails of traces of the process it is not the case that a trace is blocked by a user within a session and at the same moment within the same session a different user is blocked by the trace. If this property does not hold it can be the case that a user who submitted a Web form has to wait for an other user to interact with the application before he receives a response on his submission.

If this property holds for the initial state of an application it follows from the usage of the tails and \( tr \) functions that it holds in all possible states during the execution of the application. We can make use of a test generator and execution tool (e.g. using the TorX tools [7]) for further analysis of this and several other properties of DiCons applications. Using the example given in Section 6 will result in a set of traces which is too large to check by hand. Therefore, to illustrate, we give a small example and prove that it adheres to the independent responsiveness property.

In the following example two users can simultaneously submit an integer number via a Web form which subsequently is returned. There can be interference so that the returned number may be the number submitted by the other user.

Let \( p \) be the process defined above and \( t_0 \) be the initial time. Then \( IR((p, \lambda, t_0)) \) holds.

**proof.** Since there are only two users, \( c_1 \) and \( c_2 \), the IR property of the initial state of \( p \), \( IR((p, \lambda, t_0)) \), can be reduced to

\[
\forall \alpha \in \text{tails}(tr((p, \lambda, t))) \neg\exists k ([\text{DiCons}_{c_1}, k] \land [\text{DiCons}_{c_2}, k] \lor [\text{DiCons}_{c_2}, k] \land [\text{DiCons}_{c_1}, k])
\]

Since we make use of the merge operator \( || \), this application is (apart from the session identifier) symmetric with respect to users \( u_1 \) and \( u_2 \). So by proving that

\[
\forall \alpha \in \text{tails}(tr((p, \lambda, t_0))) \neg\exists k ([\text{DiCons}_{c_1}, k] \land [\text{DiCons}_{c_2}, k])
\]

and by making use of this symmetry we have proven \( IR((p, \lambda, t_0)) \).

Suppose that IR does not hold for \( (p, \lambda, t_0) \). Then there must be a trace \( \alpha \) in the set of possible traces \( tr((p, \lambda, t_0)) \) where a \( k \) exists such that \([\text{DiCons}_{c_1}, k] \land [\text{DiCons}_{c_2}, k] \) holds. Suppose we have an \( \alpha \) and a \( k \) such that \([\text{DiCons}_{c_1}, k] \). Then, it follows from SOS rule 31 that \( k = 0 \). This means that \([\text{DiCons}_{c_2}, 0] \), \( 0 \)

must hold. So

\[
\exists \tilde{d} \exists \tilde{y} \exists \tilde{d} (\#1((to_{c_2}(m(\tilde{d})), 0), \alpha) < \#1((req_{c_2}(\tilde{y}, \tilde{d}'), 0), \alpha))
\]

However, if there is an element \((to_{c_2}(m(\tilde{d})), 0)\) or \((req_{c_2}(\tilde{y}, \tilde{d}'), 0)\), it follows from 31 that there must be a subprocess on the left-hand side of the merge operator \( || \) that can do a step where the user is \( c_2 \). This is not the case, so we get \( \infty < \infty \) which proves false. So \( \neg[\text{DiCons}_{c_2}, 0] \). And therefore
there is no $k$ such that $\Box_{c_1,k} \land \Box_{\alpha,k}$. So $IR((p, \lambda, t_0))$ holds. \hfill \square

So by giving the semantics of $DiCons$ applications we are able to prove properties of arbitrary $DiCons$ programs. This is an important step in the future development of the $DiCons$ language and its application area.

8 Related Work

Closest to our work is the development of the Web-language $Mawl$ [1, 18]. 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. There is no formal semantics of Mawl.

Groupware [26] 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.

Visual Obliq [8] is an environment for designing, programming and running distributed, multi-user GUI applications. Its interface builder outputs code in an interpreted language called Obliq [10]. Unlike $DiCons$, an Obliq application can be distributed over several so-called sites on a number of servers. In [24], Nestmann et al. introduce Ojeblik, a distribution-free subset of Obliq together with three different configuration-style semantics for this language.

$COCA$ [20] is a generic framework for developing collaborative systems. In $COCA$, participants are divided into different roles, having different rights like in $DiCons$. For this language no formal semantics has been developed.

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 [11, 17]. The formal semantics of Service Combinators are given in [11]. A further development is the so-called ShopBot, see [13].

9 Conclusions

We have given a complete operational semantics for the behavioural part of the Internet language $DiCons$. This allows us to translate an arbitrary $DiCons$ program into a labeled transition system. In turn, such a labeled transition system can be analysed using a model checker, so that properties of $DiCons$ programs can be verified. As an example, we explained the independent responsiveness property and proved that it holds for a small $DiCons$ application.

$DiCons$ is a language that supports the development of Internet applications at the right level of abstraction. We have used the language in a number of experiments, but still need more experience in larger applications. This will point the way to some useful extensions like atomic regions (as in Mawl [1, 18]), database coupling (as in Strudel [19]) and style sheets.

We have restricted our formal treatment of Internet applications to closed systems, i.e. systems consisting of a single server and a number of users. It would be interesting to see if our approach could be extended to open systems, i.e. systems where the server can communicate to external entities, such as other servers. The set of communication primitives, as described in the current paper, does not support the description of interactions between servers.
Nevertheless, this asymmetric relation between users and servers is not essential to our approach. One could easily define for each of the DiCons interaction primitives a dual version: an active push would correspond to a passive pull, a reactive push to an active pull, etc. These dual primitives would behave more or less in the same way as the already discussed primitives. Using such additional primitives, we could specify peer-to-peer communication, e.g., a server querying another server.

We consider such an extension as an important step in the future development of the DiCons language. The main reason that we restrict ourselves, for the moment, to the asymmetric situation is that we want to gain experience with designing and analysing such simple systems first. The symmetric situation where servers communicate to each other yields systems which are much harder to analyse. The reason is that the participating servers operate in parallel. Describing this situation would require parallel composition of server behaviours and matching of corresponding interaction primitives.

We implemented a compiler to translate DiCons specifications into Java Servlets [23]. Except for generating Servlets, the compiler checks specifications with respect to syntax and static requirements. We have made a graphical representation of DiCons using Message Sequence Charts, see [3]. More information on DiCons, its compiler and some working examples can be found in [6, 4] or at http://dicons.eesi.tue.nl/.

Acknowledgements. We would like to thank Michel Reniers and Louis van Gool for their valuable comments on previous versions of this paper.

References


