Web Cube: a Programming Model for Reliable Web Applications

I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra, B. Widjaja

institute of information and computing sciences, utrecht university

technical report UU-CS-2005-002

www.cs.uu.nl

Web Cube: a Programming Model for Reliable Web Applications

I.S.W.B. Prasetya (UU), T.E.J. Vos (UPV), S.D. Swierstra (UU), B. Widjaja (UI)

nov. 2004

UU: Institute of Information and Computing Sciences, Utrecht University. P.O.Box 80.089, 3508 TB Utrecht, the Netherlands. UI: Fakultas Ilmu Komputer, Universitas Indonesia. Kampus UI Depok, Indonesia. Supported by RUTI-2 Grant. UPV: Instituto Tecnológico de Informática , Universidad Politécnica de Valencia.
Emails: wishnu@cs.uu.nl, tanja@iti.upv.es, doaitse@cs.uu.nl, bela@cs.ui.ac.id

Abstract

Web Cube is a server side programming model for building interactive web applications. Compared to typical server side approaches used nowadays, Web Cube offers better abstraction mechanism and built-in safety. It imposes a more logical organization of its components. As a direct result of this applications become inherently secure and development is less error prone. Web Cube is based on Seuss, a framework for modelling distributed systems. Seuss is rather light weight, but is abstract enough to allow convenient programming. It comes with a simple but powerful logic that allows safety and other temporal properties of a Web Cube application to be specified and verified.

Contents

```
<% if (Auth.isAdmin(user)) { %>
   <b> Welcome <% if (counter>0) { %> again <% }; %>
       <%= user.Name %> !
   </b>
<% } else { %>
   Hello stranger!
<% } %>
```

Figure 1: *A fragment of a Java Server Page (JSP).*

1 Introduction

The Internet has changed the way we deal with information. In particular, Web applications have affected our daily life in many ways, since they are being used in information management/gathering, e-commerce, e-government, banking, e-learning, entertainment, etc. This pervasive and radical growth of Web applications, and the impact this has on society, makes correctness a primary concern.

The most commonly used techniques to develop sophisticated Web applications nowadays are technologies like PHP, ASP, and servlets. These technologies, however, are still prone to errors and security abuses. This is caused by:

1. The used programming method itself. Many low level details explicitly have to be programmed by hand. It is common to write an active page by stitching program fragments into HTML pages. Although some frameworks support a quite sophisticated stitching scheme, they do not clearly define the relation between a page and the stitched code fragments that are responsible for its dynamic behaviour. Not only does this make things tricky for less experienced programmers, it also makes reasoning about the behaviour of the complete application difficult.

   Figure 1 shows a fragment of an hello-world variant written in JSP. The code is not pleasant to read (let alone to debug). There are three languages co-existing in the code (HTML, Java, and JSP itself). Although this in itself is not bad, the languages here are interleaved quite haphazardly. The full stitching scheme of JSP is even more complicated, which makes it difficult to write a decent syntax analysis tool. In fact, in JSP there is no syntax and type checking done at the JSP level; syntax errors may slip through undetected until a user bumps in to them by accessing an erroneous page for the first time.

2. Any form of verification is hard due to the absence of a formal semantic model of the used programming languages. Java shows that simply providing such a semantics does not necessarily lead to a practical solution. The full semantics of Java is very complicated and verification against it may not be feasible.

The contribution of this paper is a programming model, called *Web Cube*, for constructing interactive component-based web applications. Web Cube offers a simple but well defined concept of a web application and how it can be constructed from its components. Web Cube strives to be abstract, such that programmers, to some degree, are relieved from laborious low level code plumbing. In order to enable the specification and verification of temporal properties of a web application, we have casted Web Cube with Misra's Seuss [8]. Seuss is a formal framework for modelling distributed systems that comes with a simple and abstract logic, which can help significantly in lowering the verification cost. Web Cube logic extends Seuss logic by being component-oriented. For this, Web Cube uses an extension taken from our previous work on component-based theories [12, 11, 9] that enable abstract reasoning over potentially large and complicated resources, such as

an enterprise database. The work reported here is still on-going and there is no implementation yet.
This paper is organized as follows. Section 2 introduces Seuss. Section 3 explains how a Web Cube application is constructed and the associated execution model. Section 4 describes the semantics of a Web Cube application and Section 5 explains how properties can be verified. Section 6 discusses related work, and Section 7 concludes.

2 Seuss

In Seuss a program is called a *box*. Each box has a state; the state of a box is given by the values of the variables of that box.

A box can have a set of *action*s, that may be guarded. These actions are executed as in a Weakly Fair Transition System: an execution is an infinite sequence of steps; at each step an action is selected for execution. The selection process is non-deterministic and such that each action is selected eventually. Only when the selected action's guard evaluates to true, the action is executed, else the effect is just a skip. All actions should be terminating, and are atomic.

A box can also have *method*s. Methods can be called from within another method or from an action[1]. A method may have parameters, which are passed by value (thus avoiding aliasing issues). Web Cube will use a more restricted calling convention than Seuss [8]: (1) all methods are unguarded (total) and thus can be called from anywhere within an action or another method; (2) a method passes values back to its caller through a return value. These restrictions are just for brevity and are not essential. We will also allow a method that has no side effect to be declared as a `function`.

The body of an action or a method can be written in any language that supports Hoare logic. However, since the Seuss logic requires actions and methods to be terminating, it is the programmer responsibility to guarantee this.

Generic boxes can be defined by categories called `cat`s, and several boxes can be created from each `cat` by instantiation. Note that in this way `cat`s and boxes are analogous notions of, respectively, classes and objects in Java. Figure 2 shows an example of a category called `VoteServer`. Via the method `vote` a box of this category allows an environment to send a vote. The box has two actions. The action `move` moves the content of the incoming votes buffer to an internal buffer `h`. The action `validate` validates the votes in `h` (e.g. using some cryptographic technique); it stores the valid votes and add them to the total count. Note by the way that: (1) `move` can be implemented as an $O(1)$ operation, and (2) `validate` does not use the variable `votes`, and hence can be executed concurrently with calls to `vote` by the environment. The environment can also call the method `info` to inquire whether the voting is still open and the number of votes counted so far.

In Seuss all variables of a box are private. For convenience, however, we also allow public variables. In Web Cube, by default a variable is public, unless it is specified as private. A public variable of a box is readable from surrounding boxes (see below), but changing the value of a variable can only be done via a method.

Inner Box. We will also allow a box to contain inner boxes (this feature is not explicitly present in the original Seuss). We will only consider inner boxes which can be created statically. In principle an inner box can be declared to be private, though for brevity here we assume all inner boxes to be public. Public variables and methods of an inner box are accessible from the environment of its parent box using a qualified notation as in Java. A box P that contains inner boxes can be transformed to an equivalent *flat* box, denoted by $\overline{P}$, which is a box without inner boxes. Essentially this is done by applying some renaming scheme on P's inner boxes so that each has a unique name space.

[1]Note that since actions are atomic, calling a long series of methods from an action may lock some variables for quite a long time. The programmer should take this into account.

```
cat VoteServer {
    VoteList votes      = [] ;
    VoteList validVotes = [] ;
    bool     open       = True  ;

    private int        count = 0     ;
    private VoteList   h     = []    ;

    method vote(v:Vote) { if open then insert(v,votes) else skip } ;

    function (Int,Bool) info() { return(count,open) } ;

    method stop()  { open := False } ;

    partial action move
            null h /\ ~null votes  -> h,votes := votes,[] ;

    partial action validate
            ~null h -> { Vote v ;
                         while ~null h do
                           { v := h.getHead() ;
                             if v.isValid()
                                then { insert(v,validVotes); count:=count+1 }
                                else skip }
                       }
}
```

Figure 2: *An example of a category.*

Specification. Seuss logic has operators as unless and $\mapsto$ (leads-to) to express temporal properties. The logic is defined in terms of flat boxes —the logic is mostly the same as its predecessor UNITY [3]. For example, if x is a box, $x \vdash p$ unless q means that all actions a in $\overline{x}$ satisfy: $\{p \wedge \neg q\}\ a\ \{p \vee q\}$. Hence, if x enters a state satisfying p, it will remain in p or go over to q (later we will need to alter this definition to deal with components).

We also allow methods to be used in a predicate which in turn can be used in a specification. If p is a predicate and $m(e)$ is a call to a method m, $p; m(e)$ is a predicate specifying the set of states obtained after $m(e)$ is called on states satisfying p. It is formally defined as follows (in terms of Hoare triple):

$$p; m(e) \ = \ (\bigwedge q :: \{p\}\ m(e)\ \{q\}) \tag{1}$$

For example, if `insert` is a method of a box P, then

$$P \ \vdash \ V{>}0; \texttt{insert}(V, s) \ \mapsto \ V \in s$$

states that after a call to $\texttt{insert}(V, s)$ with a positive value V, eventually this value will be in s.

Functions of a box can also be used in a specification. For example, if f is a function, then the expression `let` $x = f(e)$ `in` $x > 0$ is a predicate specifying the set of states for which $f(e)$ returns a positive value.

3 Execution Model

A Web Cube application, or simply application, is built by composing *cubes* and *resources*. Both are Seuss boxes, i.e. programs having their own states.

A cube is a server side program that interacts with a client. The client does so by calling the cube's methods. The interaction goes via an HTTP connection. So, a user can use a web browser to send HTTP requests to a cube, which would be translated to method calls. Calls from a client are handled atomically and may alter the cube's state. The method may also send back HTML responses, which the user can view in his browser. We will call a method that produces HTML a *writer* method.

The role of a cube inside an application is purely for serving client's requests. It means that as a Seuss box it does not have any actions (it only has methods). A resource on the other hand, may have actions. Resources will be explained in more detail below.

The life time of a cube is one session. So, compared to typical CGI scripts, the state of a cube is more persistent. Also, access to cubes is strongly guarded to provide some built-in security. We will return to this issue later.

Below is a simple example of a hello-world application. The application is made of of a single cube called `home`.

```
application helloWorld {

  cube home {

     n : Int = 0 ;

     writer method home() {
       respond("hello world! <@ n @>")   ;
       n := n+1                        }
  }
}
```

Each application, like the one above, should have a cube called `home` that contains a method `home`. The method is always called automatically when an instance of the application is created (upon a user's request) and so resembles the home-page of the application. In the above case,

calling `home` will cause the user to see the string `hello world` on his browser, followed by whatever the value of `n` is at that moment. The variable `n` is increased each time the client calls the method `home` (for example if the user hits the reload button).

Between the symbols `<@` and `@>` we can embed a Seuss expressions to be inlined inside a response string. We will explain this in more detail later, but compared to for example JSP, Web Cube has a much simpler inlining scheme.

The Corridor

To make Web Cube work, we need an interface program since a standard web server would not know how to handle Web Cube specific HTTP requests. We will call the interface the *Corridor*, which is placed between HTTP ports and the cubes. One of the tasks of the Corridor is to recognize cube-specific HTTP requests coming at the ports, and to redirect them to the right cube.

Another important task of the Corridor is to automate user authentication, because this is too critical to be programmed by hand. When a user requests the creation of an instance of an application $\mathcal{A}$, he should first authenticate himself. The Corridor takes over the task of authenticating the user, which includes verifying that the user has sufficient access right to all resources required by the requested application. Web Cube also allows so-called *clearance level* to be specified. In this paper we just consider two levels: *Admin* and ordinary. Some methods of the cubes can be specified to be accessible only to the user with an Admin clearance. It is also the task of the Corridor to verify the user's clearance.

Once a user passes the authentication phase a new instance A of $\mathcal{A}$ will be deployed for him. A unique application instance ID or *ainid* and will be generated to uniquely identify communications between the user's client and A.

Packages exchanged during authentication, and subsequent communication between a client and an application instance should be properly encrypted. Since this can be adequately and transparently taken care of by for example SSL, we will abstract away from encryption in our programming model.

To call a method, the Corridor requires the client to also specify the method's *address*. The address of a method is a unique URI string that will allow the Corridor to uniquely identify the method within the set of application instances that it has access to. It will typically include the method's name and its *ainid*. How an address is constructed is not essential here.

Cube Integrity

In addition to the safe access guarding provided by the Corridor, Web Cube also has a number of other safety measures. We have said that a client interacts with a Web Cube application by calling its methods. Each method call (but not a series of them) will be handled atomically. Furthermore, HTML responses are generated without further side effect. As a consequence, to analyze the safety of an application it is sufficient to consider its methods, with all `respond` commands striped out.

With the exception of resources, applications and application instances do not share any data, and thus interference between different parts of an application (which is the typical source of errors in distributed systems) can be contained.

Web Cube does not encourage code stitching like in PHP or JSP because this is too error prone. However, for convenience some stitching is still allowed through the `<@ @>` notation to do response inlining (we have seen it in the previous example). This inlining scheme is safe as long as it is side effect free.

Resources

In Web Cube a resource represents a device, such as an enterprise database, which is typically shared across application instances or even across applications. The life time of a resource can be thought to be permanent. Essentially, a resource is just a reactive program that has its own state and may provide methods. It may also perform some computation in the background that

```
application webVote {

  resource d:MyVoteDB ;

  cube home {

    writer method home() {
      respond(
        "<form method=post action=<@address.d.vote@>>
         Enter your vote: <input type=text name=v>
         <input type="submit" value="SUBMIT">
         </form>

         <p><a href=<@address.d.info@>>Click here to get vote info</a>
         <p><a href=<@address.logout@>>logout</a>
         <p><a href=<@address.d.close@>>Close the vote (Admin only)</a>
        ") }

    writer method info() {
      n:Int  ;
      b:Bool ;
      (n,b) := status() ;
      respond("<p>Total votes = <@n@>") ;
      if b then respond("<p>Voting is still open.")
           else respond("<p>Voting is closed.")
      }
  }
}
```

Figure 3: *A simple Web Cube application for electronic voting.*

regularly updates its state. Because a resource such as a database is a large and complicated system, most likely it will not be written in Seuss. Consequently, we will represent resources as black box programs, that in [14] are also called *component*s.

All resources should have implicit `connect` methods. When a user request an instance of $\mathcal{A}$ to be created, the Corridor will use these methods to try to connect to all resources required by $\mathcal{A}$. For this to be successful, the user must have access right to all resources needed by $\mathcal{A}$. Only when all `connect`s are successful an application instance will be created.

Figure 3 shows a simple application to do electronic voting. It uses a single resource `d`, that represents a back-end database where the votes are stored. The resource supports three methods (not shown in the Figure): `vote` that is used to enter a new vote, `info` that is used to query the status of the voting, and `close` that is used to close the voting. The last is only accessible to users with an *Admin* clearance .

The first time an instance of `webVote` is created, the Corridor will connect to the database resource and the method `home` will be called. The HTML code generated by `respond` will be sent to the user's browser and he will see a simple form where he can type in his vote, a submit button, a link to get the voting status, and few other things.

Disengaging an Application

Each application always has a pre-generated `logout` method used to disengage the corresponding application instance. It will also disconnect the application instance from its associated resources. The method can be called by the user, or by the Corridor which can decide to do so for some security

reason. When an application instance is disengaged it will be disengaged entirely. Disengaged cubes can be garbage collected, but since this may not happen immediately, the Corridor must make sure that no HTTP request can reach disengaged cubes.

HTML Responses

HTML responses are generated by the `respond` method. It expects a string, which will just be sent as is to the client. Seuss expressions can be inserted within the string argument of `respond` by putting them between `<@` and `@>` as in:

```
respond{"hello <@name@> !"}
```

As `respond` encounters an embedded expression `<@`e`@>`, it will evaluate e and put the result in the place where e stands. This is called *response inlining*. In addition to ordinary Seuss expression, we also allow the following special form. Let m be a method name. An expression of the form `address.`m will be substituted by m's URI address. For example:

```
respond("<a href=<@address.home@>>back home</a>")
```

will cause a href-ed string `back home` to be displayed in the user's browser. Clicking on the link will cause the method `home` to be called (with no parameter).

Any expression can be inlined as long as it does not have side effects (forbidden; see Cube Integrity). Nesting `respond` calls via inlining is therefore not allowed (which is also a bad style). It also implies that the order in which inlined expressions within the same `respond` are evaluated is inconsequential.

If it is convenient, an implementation may also allow an `include` method for reading static HTML content of a file and sending it to the client.

Potentially, someone can still write code like this:

```
respond{"<b"}; x:=x+1; respond{"> hello </b>"}
```

which can be avoided by writing it like this:

```
x:=x+1; respond{"<b> hello </b>"}
```

Mangling can be prevented at the parser level. This does require additional work in writing the parser, but it is not much because of the simple inlining scheme adopted here.

Specifying Safety

Since cubes and applications are, in principle, just Seuss boxes, their properties can be expressed formally using the Seuss specification language. For example, the following property can be required for the `webVote` application (Figure 3):

$$\texttt{let}\ (n, -) = \texttt{d.info}()\ \texttt{in}\ n \geq N \quad \textsf{unless} \quad \textsf{false} \tag{2}$$

If `info` returns a pair, whose first element tells us how many valid votes have been counted so far, the above property essentially implies that the application will not silently cancel an already counted vote.

The Seuss logic can be used to verify a specification such as the one above. However first we need to define the semantics of cubes in terms of boxes. This is explained in the next section.

4 Semantics

Now that we have detailed the execution model of Web Cube applications, we can now look at their semantics and see how we can specify and verify properties about them. In the sequel a, b, c are actions, i and j are predicates intended to be invariants, p, q, r are predicates, P, Q, R are action systems (explained later), x, y, z are boxes. Since a box can be transformed to a semantically equivalent flat box, we will only consider the latter sort.

4.1 Preliminaries

Predicate Confinement. Predicates specify a set of program states. A predicate p is *confined* by a set of variables V (written p **conf** V) if p does not constrain the value of any variable outside V. As a rule of thumb, an expression e is confined by its own set of free variables. We write p, q **conf** V to abbreviate p **conf** V and q **conf** V.

Actions. An *action* is an atomic, terminating, and non-deterministic state transition. An action can be modelled by a function from the universe of states, denoted by State, to $\mathcal{P}(\mathsf{State})$. Actions can be (multiple) assignments or guarded actions. If V is a set of variables, $\mathsf{skip}\ V$ is an action that does not change the variables in V. Guarded actions are denoted by g `-->` a, meaning that a will be executed if g is true, otherwise the action behaves as a skip. If a and b are actions, $a \sqcup b$ is an action that either behaves as a or as b. So, $(a \sqcup b)\ s = a\ s\ \cup\ b\ s$. If Σ is a set of actions then $\sqcup\Sigma$ is a shorthand for $(\sqcup a : a \in \Sigma : a)$.

Notation. We will use tuples to represent composite structures, and selectors to select the various parts. For example, $Box = (\mathsf{main} :: ActionSys, \mathsf{meths} :: \{Method\})$ defines a type Box consisting of two-elements tuples. If (P, M) is a value of this type, then $x.\mathsf{main} = P$ and $x.\mathsf{meths} = M$.

Action System. The *action system* part of a Seuss (flat) box is the set of actions of the box. It is the part of the box that autonomously and constantly updates the box state. Formally we represent an action system by the following structure:

$$ActionSys \ \stackrel{d}{=}\ (\mathsf{acts} :: \{Action\}, \mathsf{init} :: Pred, \mathsf{pub} :: \{Var\}, \mathsf{pri} :: \{Var\}) \tag{3}$$

$P.\mathsf{init}$ is a predicate specifying P's possible initial states, $P.\mathsf{pub}$ is the set of P's public (shared) variables, and $P.\mathsf{pri}$ is the set of P's private (local) variables. We write $P.\mathsf{var}$ to refer to $P.\mathsf{pub} \cup P.\mathsf{pri}$. Implicitly, $P.\mathsf{init}$ has to be confined by $P.\mathsf{var}$; $P.\mathsf{pub}$ and $P.\mathsf{pri}$ are disjoint; and for every action $a \in P.\mathsf{acts}$, it holds that for every state s, $a\ s$ is non-empty. An action system is the same as what in [3] is called a UNITY program.

Composing two action systems means running them in parallel. The behaviour of the parallel composition of P and Q is modelled by the $P[\!]Q$ which is defined as follows:

$$P[\!]Q \ \stackrel{d}{=}\ (P.\mathsf{acts}\ \cup\ Q.\mathsf{acts},\ P.\mathsf{init} \wedge Q.\mathsf{init},\ P.\mathsf{pub}\ \cup\ Q.\mathsf{pub},\ P.\mathsf{pri}\ \cup\ Q.\mathsf{pri}) \tag{4}$$

In the above definition, we assume that the private variables of P and Q have been renamed so that their names will not clash after the composition.

Invariant. A predicate i is a *strong invariant* of an action system P, denoted by $P \vdash \mathsf{sinv}\ i$, if it holds initially, and it is maintained by every action of P.

$$P \vdash \mathsf{sinv}\ i \quad \stackrel{d}{=} \quad P.\mathsf{init} \Rightarrow i\ \wedge\ (\forall a : a \in P.\mathsf{acts} : \{i\}\ a\ \{i\}) \tag{5}$$

A predicate j is an *invariant* if there exists a strong invariant i implying j.

Refinement. We define the following notion of refinement over actions —it is a variant of the standard one, e.g. as in [2]. Let V be a set of variables, and let i be a predicate (i is intended to be an invariant). Action b *weakly refines* action a (or a is an abstraction of b) with respect to V and i, if b can either simulate whatever a can do on the variables in V, assuming i holds initially, or it skips. Formally:

$$V, i \vdash a \sqsubseteq b \quad \stackrel{d}{=} \quad (\forall p, q : p, q\ \mathbf{conf}\ V : \{i \wedge p\}\ a \sqcup \mathsf{skip}\ V\ \{q\}\ \Rightarrow\ \{i \wedge p\}\ b\ \{q\}) \tag{6}$$

We will use the following simple notion of refinement for action systems:

Definition 4.1 : Refinement/Abstraction

$$i \vdash P \sqsubseteq Q \stackrel{d}{=} \begin{array}{l} P.\mathsf{pub} \subseteq Q.\mathsf{pub} \ \wedge \ P.\mathsf{pri} \subseteq Q.\mathsf{pri} \ \wedge \ Q.\mathsf{init} \Rightarrow P.\mathsf{init} \\ \wedge \\ \forall b : b \in Q.\mathsf{acts} : P.\mathsf{var}, i \ \vdash \ \sqcup \ P.\mathsf{acts} \ \sqsubseteq \ b \end{array}$$

So, under the invariance of i, $i \vdash P \sqsubseteq Q$ means that every action of Q either does not touch the variables of P, or if it does it will not behave worse than some action of P. When $i \vdash P \sqsubseteq Q$, we call Q a refinement of P, or equivalently, say that P is an abstraction of Q.

This refinement relation is a weak one. For example, unlike [15] it does not preserve progress. However, it is still useful. It still preserves safety [11], and as we will see later, it can be made to preserve progress. Moreover, to compensate for its weak expressiveness, it is relatively easier to verify.

4.2 Properties

Properties of a box can be expressed in terms of the properties of its action system. This section discusses how these properties can be specified and what the semantics of these specifications are.

Recall that an application can refer to resources that we represent as black box components. When composing an application with such a component, we should take into account that the components complete code may not be known. The standard Seuss operators unless and ensures for specifying one-step temporal progress properties are not expressive enough for reasoning under these circumstances. Below we introduce our definitions of these operators that, as we have proven in [11], are sufficient to to deal with black box components.

Definition 4.2 : Basic Operators

1. $P, i \ \vdash \ p \ \mathsf{unless} \ q$
   $\stackrel{d}{=} P \ \vdash \ \mathsf{sinv} \ i \ \ \wedge \ \ p, q \ \mathbf{conf} \ P.\mathsf{var} \ \ \wedge \ \ (\forall a : a \in P.\mathsf{acts} : \{i \wedge p \wedge \neg q\} \ a \ \{p \vee q\})$

2. $P, i \ \vdash \ p \ \mathsf{ensures} \ q \ \stackrel{d}{=} P, i \ \vdash \ p \ \mathsf{unless} \ q \ \wedge \ (\exists a : a \in P.\mathsf{acts} : \{i \wedge p \wedge \neg q\} \ a \ \{q\})$

The general progress operator $\mapsto$ is defined as the least transitive and disjunctive closure on the relation $(\lambda \ p \ q. \ P, i \ \vdash \ p \ \mathsf{ensures} \ q)$. Progress defined in this way is still difficult to preserve when subjected to parallel composition. Essentially, this is because we do not put any constraint on the environment. Obviously no non-trivial progress property can withstand an environment that can do anything.

Consider an action system P that is composed with an environment B. Consider the progress $p \mapsto q$ in the composition $P[\!]B$. Suppose we know that this progress is driven solely by P. Now, if B is an abstraction of a concrete environment Q, then we can expect that the same property will be preserved in $P[\!]Q$. To express this kind of reasoning, we introduce a new set of extended operators, with which the notion of "progress is driven solely by P" can be specified.

Definition 4.3 : Extended Progress Operators
Let P and B be action systems. We define:

1. $P_{\triangleleft}[\!]B, \ i \ \vdash \ p \ \mathsf{ensures} \ q \stackrel{d}{=} P[\!]B, \ i \ \vdash \ p \ \mathsf{unless} \ q \ \wedge \ (\exists a : a \in P.\mathsf{acts} : \{i \wedge p \wedge \neg q\} \ a \ \{q\})$

2. $P_{\triangleleft}[\!]Q, \ i \vdash p \mapsto q$ is defined such that $(\lambda p, q. \ P_{\triangleleft}[\!]Q, \ i \vdash p \mapsto q)$ is the smallest transitive and disjunctive closure of $(\lambda p, q. \ P_{\triangleleft}[\!]Q, \ i \vdash p \ \mathsf{ensures} \ q)$.

Now we can introduce the following important theorem whose proof can be found in [11]. It states that every progress property from $p \mapsto q$ made by P, when specified in terms of $P_{\triangleleft}[\!]B$, will be preserved when P is composed with any program Q that refines B.

Theorem 4.4 : PRESERVATION OF $\mapsto$

$$\frac{P_{\triangleleft}[\![B, i \vdash p \mapsto q \quad \wedge \quad j \vdash B \sqsubseteq Q \quad \wedge \quad i \Rightarrow j}{P[\![Q, i \vdash p \mapsto q}$$

Notice that the rule's premise assumes that i is a strong invariant of P. However, if P is the action system of a black box, its contract may not reveal what i is, for example because it would expose too much internal state of the box. Fortunately, the theorem says that it is sufficient if the contract exposes a weaker invariant j, since we can infer the conclusion above by showing the refinement $B \sqsubseteq Q$ based on the weaker j.

Notice also that the theorem above implies that the simplistic notion of refinement that we have chosen, while it does not preserve progress in general, it does preserve progress in terms of $P_{\triangleleft}[\![B$.

4.3 Box

A (flat) box can be represented by an action system plus a set of methods. For brevity we will only consider public methods:

$$Box = (\mathsf{main} :: ActionSys, \mathsf{meths} :: \{Method\})$$

We will also overload the selectors used on action systems so that they also work on boxes, e.g. if x is a box, $x.\mathsf{pub}$ is equal to $x.\mathsf{main}.\mathsf{pub}$. We also overload $[\![$ and $_{\triangleleft}[\![$, e.g., $x[\![Q$ means $x.\mathsf{main}[\![Q$.

The (public) methods of a box x can be called by a surrounding box y. An action in y can even call multiple methods of x. The 'Seuss Virtual Machine' (SVM) on which x runs is responsible for scheduling the actions in such a way that each is executed atomically.

We may want to expose a box x so that its methods can be called by an external program y, also called an *external environment*, which may be a non-Seuss program running on a different machine. In particular, y may be beyond the control of x's SVM. We will therefore assume weaker synchronization with such a y. Each method call from y to x is guaranteed to be executed atomically. Atomicity over a series of calls is not guaranteed.

The worst possible external environment of x can be characterized by the following action system:

$$x.\mathsf{env} \stackrel{d}{=} (\Sigma, x.\mathsf{init}, x.\mathsf{pub}, \emptyset)$$

where Σ is a set of actions modelling all possible calls to the methods in $x.\mathsf{meths}$. We will leave out the formal definition of Σ. See for example [12]. By definition, any real or *proper* external environment Q of x is a refinement of $x.\mathsf{env}$. Formally:

$$Q \text{ is a proper external environment of } x \stackrel{d}{=} (\forall i :: x.\mathsf{pub}, i \vdash x.\mathsf{env} \sqsubseteq Q)$$

Any unless and $\mapsto$ properties proven with respect to $x[\![x.\mathsf{env}$ and $x_{\triangleleft}[\![x.\mathsf{env}$ respectively, will be preserved when x is composed with any proper external environment.

4.4 Component and Contract

Recall that a web application may have access to resources, which are assumed to be available as black box components. Rather than revealing full information about itself, a component only reveals partial information in the form of a *contract*. A contract is binding: a component is obliged to realize anything it commits in its contract. Obviously the more information a contract contains, the more properties we can infer from it. On the other hand, more information makes verification harder and the component itself less reusable. When writing a contract a developer will have to consider a reasonable balance.

A contract is essentially an abstraction of the component it binds. We already have a simple refinement relation (Definition 4.1) that preserves safety and a useful class of progress properties.

Moreover, our refinement relation gives considerable freedom in deciding how much of a box's action system we want to hide. So we are going to use this relation to be the base of component-contract relation.

We will represent the actual programs behind components with boxes. This does not mean that components have to be written in Seuss. It is sufficient if they can be modelled in Seuss.

```
contract MyVoteDB {
  VoteList votes      = [] ;
  VoteList validVotes = [] ;
  bool     open       = True  ;

  method vote(v:Vote) { if open then insert(v,votes)
                          else skip } ;

  function (Int,Bool) info() { return(validVotes.length(),open) } ;

  admin method stop()  { open := False } ;

  smodel
    partial action fetch   ~null votes -> votes := [] ;

    action count        { VoteList vs = ANY ;
                          if vs.AllisValid then validVotes.append(vs) else skip }

  inv
    !v: v in validVotes : isValid(v)

  progress
    isValid(v) /\ (N,True)=info(); vote(v)  |-->  let (N',-)=info() in N'=N+1

}
```

Figure 4: *Above is an example of a contract. For example any box of category* `VoteServer` *(Figure 2) can be shown to meet the above contract.*

Figure 4 shows an example of the type of contracts we use here. If c is a contract, we will say that a box is bound by c when the box correctly implements c. For a contract c, $c.\mathsf{impl}$ denotes any box (component) bound by c. We will represent a contract c with a tuple of the following type:

$$Contract \;=\; (\mathsf{smodel} :: Box, \mathsf{inv} :: Pred, \mathsf{progress} :: \{ProgressSpec\})$$

where $c.\mathsf{smodel}$ (safety model of c) is a box that is an abstraction of $c.\mathsf{impl}$; $c.\mathsf{inv}$ is a predicate specifying an invariant, and $c.\mathsf{progress}$ is a set of specifications in form $p \mapsto q$ specifying progress made by $c.\mathsf{impl}$. We will overload the meaning of the selectors used on boxes so that they also work on contracts. For example, if c is a contract, $c.\mathsf{pub}$ denotes the set of all c's public variables, which is just equal to $c.\mathsf{smodel.pub}$.

Another important point is that a component is a continuously running program which is well encapsulated, in such a way that programs interacting with it are treated as external environments (in the sense as in the previous subsection). The nice thing about this is that we can predict the behavior of a component's environment: it is just $c.\mathsf{env} = c.\mathsf{smodel.env}$, if c is the component's contract.

We impose that $c.\mathsf{smodel}$ has no private variables (so $c.\mathsf{smodel.pri} = \emptyset$). Moreover, $c.\mathsf{inv}$ specifies a strong invariant of $c.\mathsf{smodel}$, and of any proper (external) environment of $c.\mathsf{smodel}$. Furthermore,

$c.\mathsf{inv}$ has to be confined by $c.\mathsf{pub}$. Formally:

$$c.\mathsf{smodel}[\![c.\mathsf{env} \ \vdash \ \mathsf{sinv}\ c.\mathsf{inv} \quad \wedge \quad c.\mathsf{inv}\ \mathbf{conf}\ c.\mathsf{pub}$$

Below, we define the relation between a contract and the box that is bound by it:

Definition 4.5 : Box-Contract Relation
Let c be a contract and $x = c.\mathsf{impl}$. The relation between x and c is as follows:

1. x and $c.\mathsf{smodel}$ have the same interface. That is, $x.\mathsf{pub} = c.\mathsf{pub}$ and $x.\mathsf{meths} = c.\mathsf{meths}$.
2. There exists a predicate i such that:
   (a) i is a strong invariant of $x[\![x.\mathsf{env}$ and it implies $c.\mathsf{inv}$.
   (b) $c.\mathsf{smodel}$ is a consistent abstraction of x. More precisely: $i \ \vdash \ c.\mathsf{smodel} \sqsubseteq x$
   (c) For every specification $p \mapsto q$ in $c.\mathsf{progress}$: $x_{\triangleleft}[\![x.\mathsf{env}, i \ \vdash \ p \mapsto q$

The invariant i mentioned above is called the *concrete invariant* of x, and will be denoted by $x.\mathsf{concreteInv}$. Note that since x and c have the same set of operations, then $x.\mathsf{env} = c.\mathsf{env}$. So, any proper (external) environment of $c.\mathsf{smodel}$ is also a proper environment of x. Also note that $c.\mathsf{inv}$ is an invariant of $x[\![x.\mathsf{env}$ because the box-contract relation implies the existence of a invariant i of $x[\![x.\mathsf{env}$ implying $c.\mathsf{inv}$. However, $c.\mathsf{inv}$ is not a *strong* invariant of $x[\![x.\mathsf{env}$.

4.5 Inferring properties from composed systems

This section shows two important theorems that allow us to infer the properties of a composed system from the properties of its components. The proofs are omitted and can be found in [11].

From the box-contract relation and from Theorem 4.4, it follows that the composition of a component x with any proper environment Q will maintain all progress properties specified in R:

Theorem 4.6 : Progress Commitment
Let c be a contract and $x = c.\mathsf{impl}$. Let Q be a proper environment of x.

$$\frac{p \mapsto q \in c.\mathsf{progress}}{x_{\triangleleft}[\![Q, \ x.\mathsf{concreteInv} \ \vdash \ p \mapsto q}$$

Any unless property proven with respect to the safety model in the contract is also a property of the actual box:

Theorem 4.7 : Safety Commitment
Let c be a contract and $x = c.\mathsf{impl}$. Let Q be a proper environment of x.

$$\frac{c.\mathsf{smodel}[\![c.\mathsf{env}, c.\mathsf{inv} \ \vdash \ p \ \mathsf{unless}\ q}{x[\![Q, x.\mathsf{concreteInv} \ \vdash \ p \ \mathsf{unless}\ q}$$

4.6 Application

Strictly speaking, an 'application' is different from its instance. Like a class in Java, an application has to be instantiated first to create a running program. For the semantics described here, such a detail is not essential; so, for convenience we will just use both terms interchangeably here. We will also abstract away from clearance level. Clearance level is a static aspect.

In the Web Cube setup, resources actually live in a special name space which is globally available to all cubes. In effect, we can actually pull them out from a web cube application, and treat them as root level entities. After pulling out the resources, an application is basically just a cube, which in turns is just a Seuss box.

Web applications are composed of cubes and resources. As indicated the resources are represented by contracts. Cubes are just boxes with no actions. The entire cube structure of an application can be equivalently represented by single flat box, which in turn can be represented by a contract of which the `smodel.main.acts`, `inv`, and `progress` sections will remain empty. Additionally, we change the cube's private variables to public (because a contract, as is now, cannot have private variables). So, we will represent a Web application by a set of contracts:

$$App \stackrel{d}{=} \{Contract\}$$

The concrete program induced by an application A is just the parallel composition of all its components (which are Seuss boxes):

$$A.\mathsf{impl} \stackrel{d}{=} (\llbracket c : c \in A : c.\mathsf{impl})$$

The worst environment of a Web application can be modelled by a client that tries all possible calls to the public methods:

Definition 4.8 : Application's Abstract Environment

$$A.\mathsf{env} \stackrel{d}{=} (\llbracket c : c \in A : c.\mathsf{env})$$

The abstract model of the whole application, denoted by $A.\mathsf{smodel}$, is the composition of the safety models exposed by the contracts of its components and the client:

Definition 4.9 : Abstract Model of Application

$$A.\mathsf{smodel} \stackrel{d}{=} (\llbracket c : c \in A : c.\mathsf{smodel}) \llbracket A.\mathsf{env}$$

The notation $A.\mathsf{inv}$ refers to the conjunction of the invariants specified by the contracts in A. Similarly, we define $A.\mathsf{concreteInv}$:

Definition 4.10 : Application's Invariants

$$\begin{array}{lll} A.\mathsf{inv} & \stackrel{d}{=} & (\bigwedge x : x \in A : x.\mathsf{contract.inv}) \\ A.\mathsf{concreteInv} & \stackrel{d}{=} & (\bigwedge x : x \in A : x.\mathsf{concreteInv}) \end{array}$$

4.7 Inferring an applications properties

The theorems from Subsection 4.5 can now be lifted to the application level to infer the properties of an application from the contracts of its components. The theorems are adapted from [12], for the proofs the reader is referred to [11]. The first theorem says that a safety property proven with respect to the abstract model of an application will extend to the concrete application itself, under any behaviour of the client.

Theorem 4.11 : Safety by Abstract Model

$$\frac{A.\mathsf{smodel},\ A.\mathsf{inv} \ \vdash\ p \ \mathsf{unless}\ q}{A.\mathsf{impl} \llbracket A.\mathsf{env},\ A.\mathsf{concreteInv} \ \vdash\ p \ \mathsf{unless}\ q}$$

Any progress property committed in the contract of any component (in particular, by a resource) in an application will be preserved by the application and the client:

Theorem 4.12 : Progress by Contract
Let c be a contract in A.

$$\frac{p \mapsto q \ \in\ c.\mathsf{progress}}{A.\mathsf{impl}_{\triangleleft} \llbracket A.\mathsf{env},\ A.\mathsf{concreteInv} \ \vdash\ p \mapsto q}$$

5 Verification

To prove safety and progress properties we use Seuss logic [8], which is not shown here. Although we have changed the definitions of Seuss temporal operators, it will be easy to verify that they maintain all basic Seuss laws using our our general proof theory in [10]. The theorems mentioned in sections 4.5 and 4.7 are important additions to the Seuss logic since they enable component-oriented reasoning on web cube applications. For example, recall property (2) of the `webVote` application from Figure 3:

$$\texttt{let}\ (n, -) = \texttt{d.info}()\ \texttt{in}\ n \geq N \quad \textsf{unless} \quad \textsf{false} \tag{7}$$

It is an important safety property, since it states that the application will not silently cancel an already counted vote. In order to verify its correctness, Theorem 4.11 indicates that it is sufficient to do so against the abstract model of the application. This significantly reduces the effort of verification.

The amount of information revealed by a contract determines which properties can be verified. For example, the contract `MyVoteDB` as in Figure 4 implies, by Theorem 4.12, that a valid vote submitted while `open` is true will eventually be counted. However, we cannot verify that no fake votes will be inserted into the database since the resource does not commit to this in its contract.

6 Related Work

In this section we describe some of the work related to ours. In [13] a petrinet-based framework is proposed to develop web applications. The approach is refinement based and can produce a final model which can be implemented in Java. [5] proposes a small language cl to specify a conversation between two web applications. States cannot be specified in cl. Its purpose is specifically to specify conversation protocols. Our approach is complementary to the two approaches described above. Like [13] we also use refinement, but its role is to bind contracts. We do not use refinement in the development. In our framework, the cubes are immediately executable; the development time is thus shorter, but post-verification is needed. The work in [13, 5] is more suitable to treat tightly synchronized distributed applications which are somehow deployed over HTTP connections. In the case of [13] applets are used, and [5] deals with coordination between multiple applications. In contrast, our framework tends more towards a client/server scheme.

[6] uses communicating finite automata to model web applications. Temporal properties can be specified and verified using standard temporal logics. It is however a verification approach, not a development approach. Models are incrementally and automatically constructed from intercepted HTTP packages and then subsequently subjected to verification against some pre-specified properties.

There does also exist work seeking to apply formal methods on other aspects of web applications. We only mention some of them: [7] offers a formal model for web queries; [4] offers a client-side logic programming model for web pages; [1] describes a rewriting based framework for the automated verification of Web sites which can be used to specify integrity conditions for a given Web site, and then automatically check whether these conditions are fulfilled.

The Semantics Web is an approach that nowadays draws a lot of attention. It proposes standards to specify relations between different fractions of a document so that they can be subjected to analysis or transformation by tools.

7 Conclusion

We have described an alternative programming model to construct interactive web applications. It is a modest model, in particular if compared to popular approaches such as servlets and ASP. But in return it offers better security and a logic to specify and verify critical temporal properties of an application. The logic is simple. It allows abstract reasoning over potentially complicated

resources. It also allows the presentation aspect (the HTML parts) to be ignored during the verification.

The logic is not well suited to reason about the presentation part. For such a purpose, it should be complemented with approaches such as Semantic Web.

The Web Cube project is still on-going. There is no implementation yet, so we cannot at the moment say much about practical experience.

References

[1] M. Alpuente, D. Ballis, and M. Falaschi. A rewriting-based framework for web sites verification. In *5th Int'l Workshop on Rule-based Programming RULE*. Elsevier Science, 2004.

[2] R.J.R. Back and J. Von Wright. Refinement calculus, part I: Sequential non-deterministic programs. *Lecture Notes of Computer Science*, 430:42–66, 1989.

[3] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison-Wesley Publishing Company, Inc., 1988.

[4] A. Davison and S. Loke. Logicweb: Enhancing the web with logic programming, 1996. available at `www.cs.mu.oz.au/swloke/papers/lw.ps.gz`.

[5] Svend Frolund and Kannan Govindarajan. cl: A language for formally defining web services interactions. Technical Report HPL-2003-208, Hewlett Packard Laboratories, October 10 2003.

[6] May Haydar. Formal framework for automated analysis and verification of web-based applications. In *19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 410–413. IEEE Computer Society, 2004.

[7] Alberto O. Mendelzon and Tova Milo. Formal models of Web queries. In ACM, editor, *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12–14, 1997, Tucson, Arizona*, pages 134–143, New York, NY 10036, USA, 1997. ACM Press.

[8] J. Misra. *A Discipline of Multiprogramming*. Springer-Verlag, 2001.

[9] I.S.W.B. Prasetya, S.D. Swierstra, and B. Widjaja. Component-wise formal approach to design distributed systems. Technical Report UU-CS-2000-01, Inst. of Information and Comp. Science, Utrecht Univ., 2000. Download: `www.cs.uu.nl/staff/wishnu.html`.

[10] I.S.W.B Prasetya, T.E.J. Vos, A. Azurat, and S.D. Swierstra. !UNITY: A HOL theory of general UNITY. In D. Basin and B. Wolff, editors, *Emerging Trends Proceedings of 16th International Conference, Theorem Proving in Higher Order Logics (TPHOL)*, pages 159–176, 2003. Also available as tech. report No. 187 of Inst. fur Inf., Albert-Ludwig-Univ. Freiburg. Available on-line at `http://www.informatik.uni-freiburg.de/tr`.

[11] I.S.W.B Prasetya, T.E.J. Vos, A. Azurat, and S.D. Swierstra. A unity-based framework towards component based systems. Technical Report UU-CS-2003-043, Inst. of Information and Comp. Science, Utrecht Univ., 2003. Download: `www.cs.uu.nl/staff/wishnu.html`.

[12] I.S.W.B Prasetya, T.E.J. Vos, A. Azurat, and S.D. Swierstra. A unity-based framework towards component based systems. *Proceeding of OPODIS 2004: 8th International Conference on Principles of Distributed Systems*, 2004. to appear in LNCS.

[13] Giovanna Di Marzo Serugendo and Nicolas Guelfi. Formal development of java based web parallel applications. In *Proceedings of the Hawai International Conference on System Sciences, 1998*, 1998. Also available as Technical Report EPFL-DI No 97/248.

[14] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[15] T.E.J. Vos, S.D. Swierstra, and I.S.W.B Prasetya. Yet another program refinement relation. In *International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, 2002.