Web Cube

I.S.W.B. Prasetya[1], T.E.J. Vos[2,⋆], and S.D. Swierstra[1]

[1] Dept. of Inf. and Comp. Sciences, Utrecht University, the Netherlands
[2] Instituto Tecnológico de Informática, Valencia, Spain

Abstract. This paper introduces a refinement of Misra's Seuss logic, called Web Cube, that provides a model for programming and reasoning over web applications. It features black box composition of web services so that services offered by large systems, such as that of a back-end database, can be treated abstractly and consistently. It inherits the light weight feature of Seuss, which relies on an abstract view towards concurrency control, which leads to a less error-prone style of distributed programming, backed by a clean logic.

1 Introduction

Nowadays, sophisticated web applications are built using technologies like PHP, ASP, and servlets. Most are built by directly implementing them over these technologies, resulting in implementations where it is hard to separate implementation details from the core design problems. Debugging, let alone verification, is in general very hard. This is not a good practice. In theory, it is better to first design an application in an abstract-level modelling language. This is the development sequence that we will assume in this paper. At the design level, verifying critical properties is still feasible. Once verified, the design can be implemented. Subsequently, a more practical method, e.g. testing, can be used to validate the consistency between the implementation and the design. Web Cube is a *programming model*, which means it provides useful concepts and structures for constructing models of web applications and specify their critical properties. It also comes with a logic to verify a model against its properties. Web Cube is based on Misra's formalism for distributed and concurrent systems called Seuss [14]. As a modelling language Seuss is quite generic. Web Cube is more concrete than Seuss. It provides concepts which are quite specific for the domain of web applications, so that a Web Cube model can be implemented more directly.

This paper explains Web Cube's concepts and the semantics of its black box logic, which is its strongest feature. We do not at the moment offer a public implementation of Web Cube. There is a prototype, implemented by translating Web Cube source to Web Function library [10] written in the functional language Haskell. It is worth mentioning that alternatively it is often possible to implement a domain specific language by *embedding* it in a general purpose language, e.g. as the embedding of financial contract combinators in Haskell [11]. One could envisage a similar implementation of Web Cube in Haskell or in

⋆ This work has been partially supported by the Generalitat Valenciana ref. GV05/261.

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 77–92, 2006.

Java. An important benefit of embedding is that it gives a first class access to the modelling framework from the same programming language that one uses to write the application itself. This may help encouraging programmers to construct designs.

In Web Cube, a web *application* is modelled by a set of passive Seuss programs called *cubes* whose task is to coordinate a set of *web services* and interface them to users. Figure 1 shows an example of a Web Cube model of a simplified web-based voting application —more will be said at the end of Section 3. As in Seuss, we can specify temporal properties like: a valid vote submitted to the `webVote` application in Figure 1 *will* eventually be counted, or that the application never silently cancels a valid vote. Web Cube treats services as black boxes. This sacrifices completeness, but allows an application to be verified in isolation! —that is, without using the services' source code. Indeed, abstraction is now forced. But on the other hand, verification is also more feasible. Black box reasoning is however also fundamentally difficult, because parallel composition typically destroys progress properties of a component. Web Cube uses the theory from [18,19] to get a reasonably powerful black box logic while remaining light weight.

Contribution. Web Cube proposes a formal programming model for web applications with a Seuss logic support and the black box enhancement from our previous work [18,19]. With respect to Seuss we contribute an extension, namely the notions of web application and service. With respect to [18,19] the novelty here is in showing its application in the domain of web programming.

Paper Overview. Section 2 briefly introduces Seuss. Section 3 explains Web Cube's concepts and computation model. The formal machinery is described in Sections 4. Section 5 presents its black box logic. Section 6 discusses related work.

```
application webVote{
 service r = VoteServer ;  -- its contract in in Fig.4

 cube home {
   method home() {
    respond("<form method=post action=<@address.vote@>>
           Enter your vote: <input type=text name=v>
           <input type="submit" value="SUBMIT"> </form>
           <p><a href=<@address.info@>>Click here to get vote info</a>")}
   method vote(v) { r.vote(v) }
   method info() {
     var n = r.info() ;
     respond("<p>Total votes = <@n@>") ;
     if r.open then respond("<p>Open.") else respond("<p>Closed.") }   }}
```

Fig. 1. A simple Web Cube application for electronic voting

2 Seuss

In Seuss a *box* describes a *reactive* (non-terminating) program. It consists of a set of variables, a set of atomic guarded actions, and a set of methods. The variables define the state of the box. The execution of the box consists of an infinite sequence of steps; at each step an action is non-deterministically, but weakly fair, selected for execution. If the selected action's guard evaluates to true, the action is fully executed, else the effect is just a skip. The methods play a different role. They form the only mechanism for the environment to alter the state of the box. Methods may have parameters, actions can not.

For brevity we will not discuss the 'category' (generic box) [14]. When declaring a variable we omit its type specification. We only consider non-blocking methods (called *total* in [14]). Parameters are passed by value, and a method may return a value.

Figure 2 shows an example of a box called `VoteServer` —for now just consider it to have nothing to do with the `webVote` application in Figure 1. The notation `[]` denotes the empty list. The method `vote` allows the environment to submit a vote to the box. The box continuously executes one of its two actions: `move` and `validate`. The notation g `->` S denotes a guarded (and atomic) action: g is the guard, and S is the action to perform if the guard evaluates to true. The action `move` swaps the entire content of the incoming vote-buffer (`in`) to an internal buffer `tmp` —it can only do so if `tmp` is empty. The full code of `validate` is not displayed; the action takes the votes from `tmp` and if they are valid votes, moves them to `votes`; otherwise they are discarded. The environment can also call the method `info` to inquire after the number of (valid) votes registered so far.

Seuss' key feature is its abstract view towards multiprogramming. To program the control flow it only offers, essentially, parallel composition and guarded actions. It advocates that a programmer should only be concerned with specifying, essentially, the set of concurrent atomic tasks that constitute a system. He *should not be concerned with how to schedule these tasks for optimal performance* —the compiler and the run time system should figure this out. This abstract view leads to a simple logic and clean designs.

Seuss is the evolution of UNITY [4]. With respect to UNITY, Seuss adds methods (which are used to limit interference by the environment of a system) and the ability to structure a system's architecture as a hierarchy of boxes.

```
box VoteServer {
    var    in, votes, tmp = []   ;
           open = True           ;
    method vote(v) { if open then in := insert(v,in) else skip }
           info()  { return length(votes) }
           stop()  { open := False }
    action move     ::  null tmp      -> tmp,in := in,[] ;
           validate ::  not(null tmp) -> ...                  }
```

Fig. 2. *An example of a Seuss box*

Seuss logic uses a slightly different set of operators. We will stick to the old UNITY unless and $\mapsto$ (leads-to) to specify temporal properties, which are derived operators in Seuss. We will defer their discussion until Section 4 where we alter their standard definition to deal with 'components'.

3 Web Cube

A Web Cube application, or simply *application*, models a server side program that interacts with a client via an HTTP connection. Figure 3-left shows the architecture of a hypothetical Web Cube execution system. Applications (e.g. A_1, A_2, B_1, ...) may run different machines. We assume a Web Cube aware HTTP server which can direct a client's requests to the correct application, collect the application's reply, and forward it back to the client.

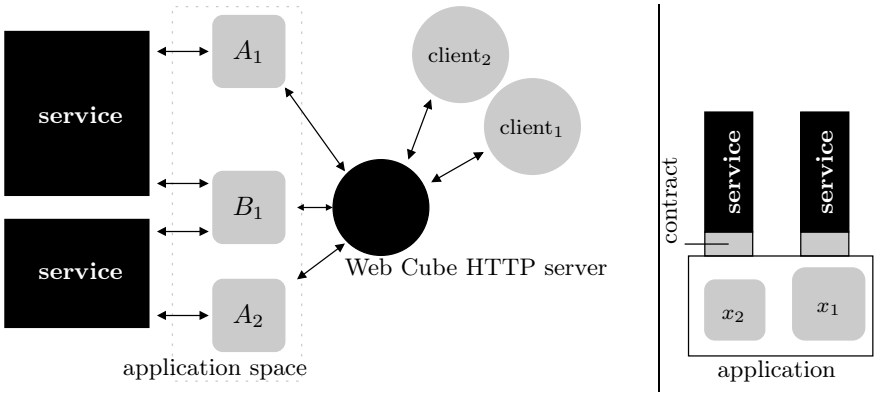

Fig. 3. Web Cube architecture

An application A is built, like in Figure 3-right, by composing so-called *cubes* (x_1 and x_2) and *web services*. A cube x is a 'passive' Seuss box —we will say more on this later— that models a part of A that can interact with a client. A client E interacts by calling x's methods. In practice this will be encoded as HTTP requests; x replies by sending back HTML responses. A web service, or simply *service*, is a black box program described by a contract and can be remotely interacted to (in practice this may happen over a SOAP/HTTP connection) by cubes. We further assume that a service is a state-persistent reactive system. Note that by attaching a contract to A it also becomes a service. A user can use his web browser to enact A's client E and interact with one of its cubes; the browser will display the cube's responses. However, E does not have to be a web browser: it can be another application using A as a service.

The role of a cube is purely for computing the responses to client's requests. It does not have reactive behavior of its own; so, we describe it by a passive Seuss box, which is a box with an empty set of actions. A service on the other hand,

may spontaneously execute actions to update its own state. Each client's request may trigger a coordinated computation over the services. For safety reason, the client can only interact with A's cubes; it cannot interact directly with A's services. So, the cubes can also be seen as providing a layer for orchestrating the services. An orchestration language, e.g. [15], can be used in conjunction to Seuss for convenient coding of the orchestration, taking into account the atomicity restriction demanded by Web Cube (Subsection 4.5).

Since a service such as a corporate database is actually a large and complicated system, we will view it as a black box specified by a *contract*. Such a contract includes a Seuss box that abstractly (thus incompletely) specifies how the service behaves as a reactive system. As in *design by contract* [13], some party, e.g. the service owner, is assumed to guarantee the consistency between the service's implementation and its contract.

An application can be deployed as a state-persistent program serving multiple clients. Another scheme, as common in e.g. servlets, is to create a fresh and exclusive instance which lasts for a single session. We are not going to make the distinction in our formal model. With respect to single-session applications, a session is treated to last infinitely long, so the application can be treated in the same way as a persistent application.

Example. Figure 1 shows a simple Web Cube application, called `webVote`, which provides an electronic voting service. It consists of a single cube called `home` and a service symbolically called `r` which is linked to the component `VoteServer` from Figure 2. Each application should have a cube called `home` that contains a method `home`. The method is called automatically when an instance of the application is created and so resembles the home-page of the application. For the `webVote` application this will cause the user's browser to show a simple form where the voter can type in his vote, a submit button, and a link to get the voting status.

HTML Responses. A cube's method m responds to a client's call by sending back its return value, encoded in HTML. Like in Java servlets, m can also generate responses by calling the `respond` method: it takes a HTML-code which will be sent to the client. The entire response of m to a call consists of the concatenation of strings produced by all the calls to `respond` in m, followed by m's HTML-encoded return value. A Web browser client may choose to display both; Web Cube applications acting as clients can only use the return value.

Responses from `respond` are however *ignored* by our Seuss semantics, which makes reasoning simpler. To do so safely we have to require that inlined expressions (below) do *not* have side effects (which is not imposed in servlets).

As in servlets, inlined expressions are allowed, as in: `respond "hello <@` e `@>"`; e will be evaluated and its result is inserted in the place where e appears. Inlined expression of the form `address.`m will be substituted by m's URI address, causing m to be called when the user clicks on it.

4 Semantics

We have explained the building blocks of a Web Cube application and its execution model. We now give its semantics, operators for specifying properties, and an extension to the Seuss logic for proving properties.

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.

4.1 Preliminaries

Selector. We use tuples to represent composite structures, and selectors to select the various parts. For example, $T = (\mathsf{a} :: U, \mathsf{b} :: V)$ defines a type T consisting of pairs whose elements are of type U and V. If $t = (u, v)$ is a value of type T, then $t.\mathsf{a} = u$ and $t.\mathsf{b} = v$.

Actions. An *action* is an atomic, terminating, and non-deterministic state transition. We model it by a function from the universe of states, denoted by State, to $\mathcal{P}(\mathsf{State})$. Guarded actions are denoted by g -> S, meaning that S will be only executed if g is true, otherwise the action behaves as a skip —the latter implies that in our model $a\ s \neq$, for any action a and state s. 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 A is a set of actions then $\sqcup A$ denotes $(\sqcup a : a \in A : a)$. If V is a set of variables, $\mathsf{skip}\ V$ is an action that does not change the variables in V, but may change variables outside V. The notation $\{p\}\ a\ \{q\}$ denotes a Hoare triple over an action a with p and q as pre- and post-condition.

Predicate Confinement. State predicates specify a set of program states. A predicate p is *confined* by a set of variables V, written p **conf** V, if p can only be falsified by actions that manipulate variables in V (it follows that p is confined by its set of free variables). We write p, q **conf** V to abbreviate p **conf** V and q **conf** V.

4.2 More Preliminaries: Box and Property

The methods of a (Seuss) box x only define the interface with which the environment interacts with x. If we strip the methods we obtain the description of the box's own program. This stripped box is called the *action system* and corresponds to a UNITY program ([4], Seuss predecessor). For conciseness we only define properties and parallel composition at the action system level, since this is sufficient for presenting our theorems later. Technically, these notions can be lifted quite naturally to the box and application level. Formally, we will represent box and action system as follows.

$$Box \ \stackrel{d}{=}\ (\mathsf{main} :: ActionSys, \mathsf{meths} :: \{Method\}) \tag{1}$$

$$ActionSys \ \stackrel{d}{=}\ (\mathsf{acts} :: \{Action\}, \mathsf{init} :: Pred, \mathsf{var} :: \{Var\}) \tag{2}$$

If P is an action system, $P.\mathsf{init}$ is a predicate specifying P's possible initial states, $P.\mathsf{var}$ is the set of P's variables. Implicitly, $P.\mathsf{init}$ has to be confined by $P.\mathsf{var}$. We will overload action system's selectors so that they also work on boxes, e.g. $x.\mathsf{var}$ means $x.\mathsf{main}.\mathsf{var}$.

A useful property is that of invariant, because it confines the set of states reachable by a reactive program. A predicate i is a *strong invariant* of an action system P, denoted by $P \vdash \mathsf{sinv}\ i$, if it holds initially, and is maintained by every action in P:

$$P \vdash \mathsf{sinv}\ i \quad \stackrel{d}{=} \quad P.\mathsf{init} \Rightarrow i \ \wedge\ (\forall a : a \in P.\mathsf{acts} : \{i\}\ a\ \{i\}) \tag{3}$$

A predicate j is an *invariant* if there exists a strong invariant i implying j. For specifying a broader range of safety properties, Seuss offers the unless operator. Let p and q be state predicates. When $p\ \mathsf{unless}\ q$ holds in P, this means, intuitively, that each action in P will go from *any* state in p to some state in $p \vee q$. Note that the definition quantifies over *all* states. We will deviate from this definition. We parameterize the property with an invariant (i), as in [20], so that the quantification over states can be restricted to those which are actually reachable by P. Moreover, we require that p and q to be confined by $P.\mathsf{var}$. Although this seems more restrictive, it does not really limit the way in which we usually use the operator. Technically, it makes the property more robust in parallel compositions [16]. Together with the definition of unless we also give the corresponding $\mathsf{ensures}$ operator, which specifies progress from p to q by executing a single action:

Def. 1 : 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 usually defined as the least transitive and disjunctive closure of $\mathsf{ensures}$. Unfortunately, progress defined in this way is difficult to preserve when subjected to parallel composition —essentially, because we do not put any constraint on the environment. We will return to this issue in Section 4.4.

We only introduce one sort of program composition, namely *parallel composition*. If P and Q are action systems, $P[\!]Q$ denotes an action system that models the parallel execution of P and Q:

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

If x and y are two boxes, we also write $x[\!]P$ to denote $x.\mathsf{main}[\!]P$ and $x[\!]y$ to denote $x.\mathsf{main}[\!]y.\mathsf{main}$.

4.3 The Underlying Component Based Approach

Web Cube assumes services to be available as black box entities, also called *components* [25]. A component only reveals partial information about itself in the

form of a *contract*. In particular, it does not reveal its full code. The component owner guarantees the consistency of the contract. Obviously a contract that reveals more information allows stronger properties to be inferred from it. However, such a contract is also more constraining, hence making the component less reusable, and the verification of the the component's implementation more costly. Consequently, when writing a contract, a developer will have to consider a reasonable balance.

Essentially the relation between a component x and its contract c is a *refinement/abstraction* relation. That is, x has to refine c (or conversely, c is an abstraction of x), usually denoted by $c \sqsubseteq x$. Such a relation preserves properties of interest: a property ϕ inferred from the contract c is also a property of the component x. In sequential programming refinement traditionally means reduction of non-determinism [2]. Lifted to distributed programming $c \sqsubseteq x$ means that every observable execution trace of x is allowed by c. This relation does not however preserve progress properties. There are a number of stronger (more restrictive) alternatives, e.g. Udink's [26] and Vos' [27], that preserve progress; but these are expensive to verify. For Web Cube, we choose a weak notion of refinement, taken from our previous work [18]. It is even weaker than simple reduction of non-determinism, and thus has the advantage that it is less restrictive, and hence easier to verify. Like most refinement relations, it still preserves safety, but surprisingly it also preserves a class of progress properties as we will see below. Although the class is much smaller than for example Vos' [27], we believe it is still quite useful.

We start by defining the refinement at the action level. Let a and b be two actions. Traditionally, $a \sqsubseteq b$ means that b can simulate whatever a can do ([2]). However, this is a bit too restrictive in the context of an action system. Imagine that a is part of an action system P, then we can ignore what b does on variables outside $P.\mathsf{var}$ or what its effect is on the states that are not reachable by P. Furthermore, we can also allow b to do nothing, since doing nothing will not break any safety properties of a. We capture these issues in our refinement relation in the following formalization. Let V be a set of variables (intended to be $P.\mathsf{var}$), and i be a predicate (intended to be an invariant, thus specifying P's reachable states). Action b *weakly refines* action a with respect to V and i is defined as follows:

$$\begin{array}{l} V, i \vdash a \sqsubseteq b \\ \stackrel{d}{=} \quad (\forall p, q :\ p, q \ \mathbf{conf}\ V :\ \{i \wedge p\}\ a \sqcup \mathsf{skip}\ V\ \{q\} \quad \Rightarrow \quad \{i \wedge p\}\ b\ \{q\}) \end{array} \tag{5}$$

Lifting this definition to the action system level gives us:

Def. 2 : Refinement/Abstraction

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

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. Parallel composition is $\sqsubseteq$-monotonic in both its arguments.

4.4 Basic Results on Black Box Composition

Like in [3,26,27] the above refinement relation preserves safety but in general not progress. However, consider the following restricted class of progress properties. Let B be an environment for P. We write:

$$P_{\triangleleft}[\![B,\ i\ \vdash\ p \mapsto q$$

to express that under the invariance of i, the composed system $P[\![B$ can progress from p to q. Moreover, this progress is *driven by* P. That is, the progress is realized even if B does nothing:

Def. 3 : Extended Progress Operators

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

The result from [18] below states that progress 'driven by x' is preserved by weak refinement over B:

Thm. 4 : Preservation of $\mapsto$

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

Note that the same does not hold for weak refinement over x.

Proof: The formal proof is by induction over $\mapsto$; we refer to [18]. Informally: assume i as an invariant of $x[\![Q$. Since $i \Rightarrow j$, j is also an invariant. Since Q refines B under j, throughout the computation of $x[\![Q$ every action of Q behaves, with respect to variables of x and Q, as some action of B or as a skip. Consequently Q cannot destroy any progress in terms of $x_{\triangleleft}[\![B$, since this progress is driven by x and cannot be destroyed by any action of B. □

The theorem below states that our notion of weak refinement also preserves safety. We refer to [18] for the proof.

Thm. 5 : Preservation of unless

$$\frac{x, i \vdash p\ \mathsf{unless}\ q \quad \wedge \quad j \vdash x \sqsubseteq x' \quad \wedge \quad i \Rightarrow j}{x', i \vdash p\ \mathsf{unless}\ q}$$

4.5 Web Cube Atomicity Restriction

Let y be the environment of a box x in a parallel composition. Seuss allows methods and actions of y to call x's methods. In particular, this allows y to perform multiple method calls to one or more boxes in a single action. Since actions are atomic, this effectively empowers y to force an arbitrary level of atomicity on its accesses to x. This is a very powerful feature, but unfortunately

it will also allow y to behave more destructively with respect to x's temporal properties. For this reason in Web Cube we will limit the feature, and define a notion of *worst allowed environment* as follows:

$$x.\mathsf{env} \stackrel{d}{=} (\{\sqcup m \mid m \in x.\mathsf{meth}\},\ x.\mathsf{init},\ x.\mathsf{var}) \tag{6}$$

where $\sqcup m$ is an action modeling the disjunction of all possible *single* calls to m. So, if m is a 1-arity method, then $\sqcup m = (\sqcup v :: m(v))$.

Now, we define a box y to be a *proper (allowed) environment* of x under an invariant i if it refines the worst allowed environment of x. More precisely:

$$y \text{ is a proper environment of } x \text{ (under } i) \stackrel{d}{=} \quad i \vdash x.\mathsf{env} \sqsubseteq y.\mathsf{main} \sqcup y.\mathsf{env} \tag{7}$$

Intuitively, every action and method of x's proper environment y can only contain a single call to a state-altering method of x. This can be checked statically. The action (method) can however still contain an arbitrary number of calls to x's functional methods (i.e. methods that do not alter the state) and calls to other boxes' methods. The proper environment condition enforces a more deterministic environment, but in return it will behave less destructively with respect to x.

4.6 Contracts

We will use the following structure to represent contracts:

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

If c is a contract, $c.\mathsf{impl}$ denotes a Seuss box which is a component associated with c. Let $x = c.\mathsf{impl}$. The methods of $c.\mathsf{smodel}$ specify the visible interface of x. The action system of $c.\mathsf{smodel}$ specifies an abstraction over x, in the sense of Def. 2. The inv section specifies an invariant. In the progress section we specify the component's critical progress properties. Only progress 'driven by' the component, in the sense of Def. 3, can be specified, so that we can use Thm. 4 to infer its preservation. In practice a component like a database is not written in Seuss. However, as long as its owner can produce the above form of contract, and guarantee it, we can proceed. The relation between c and $c.\mathsf{impl}$ is formalized by:

Def. 6 : Box-Contract Relation
If c is a contract and $x = c.\mathsf{impl}$, there should exist a predicate i such that:

1. i is a strong invariant of $x[\![x.\mathsf{env}$ and it implies $c.\mathsf{inv}$.
2. c and x have a 'compatible' interface. For brevity, here it means that both specify exactly the same set of methods: $c.\mathsf{smodel.meth} = x.\mathsf{meth}$.
3. $c.\mathsf{smodel}$ is a consistent abstraction of x, i.e. $i \vdash c.\mathsf{smodel.main} \sqsubseteq x.\mathsf{main}$
4. for every specification $p \mapsto q$ in $c.\mathsf{progress}$ we have $x_{\triangleleft}[\![x.\mathsf{env}, i \vdash p \mapsto q$.

```
contract VoteServer {
  smodel
    var    in, votes  = []    ;
           open       = True  ;
    method vote(v) { if open then in := insert(v,in) else skip } ;
           info()  { return length(votes)) }                     ;
           stop()  { open := False         }                     ;
    action fetch :: in := [] ;
           count :: {var v ;
                     if isValid(v) then votes := insert(v,votes) else skip }

  inv         v in votes ==> isValid(v)

  progress    isValid(v)/\open ; vote(v)  |-->  v in votes
}
```

Fig. 4. A contract for the component `VoteServer` (Figure 2)

The invariant i mentioned above is called the *concrete invariant* of x, and will be denoted by $x.\mathsf{concreteInv}$. This concrete invariant i is partially specified by $c.\mathsf{inv}$, since $i \Rightarrow c.\mathsf{inv}$. Its full details cannot be inferred from the contract though. The first condition above also implies that $i.\mathsf{inv}$ is an invariant of $x[\![x.\mathsf{env}$, though in general it is *not* a *strong* invariant of $x[\![x.\mathsf{env}$.

The above definition of 'compatible interface' implies $c.\mathsf{impl.env} = c.\mathsf{smodel.env}$ So, any environment which is proper according to a contract c is automatically also a proper environment of $c.\mathsf{impl}$. Actually, it would be sufficient to require $c.\mathsf{impl.env} \sqsubseteq c.\mathsf{smodel.env}$ such that we can weaken the definition of 'compatible interface' and make it more realistic. This, however, is outside the scope of this paper.

Figure 4 shows an example of a contract, that could belong to the component `VoteServer` in Figure 2. Free variables in the inv and progress sections are assumed to be universally quantified. The contract's action part reveals that `VoteServer` may from time to time empty the incoming buffer `in`. It does not, however, specify when exactly this will happen. The contract also says that the server will only fill `votes` with *valid* votes though it leaves unspecified as to where these votes should come from. Although a very weak one can infer a critical safety property from this abstraction: *no invalid vote will be included in the counting.*

For convenience, we allow methods to be used when specifying state predicates within a temporal specification in the following way. If p is a state predicate and $m(e)$ is a call to a method m, the predicate $p; m(e)$ specifies the set of states that result from executing $m(e)$ on states satisfying p. So, the progress section in Figure 4 states that after a valid vote is successfully submitted (which only happens if `open` is true) through a call to the method `vote`, eventually the vote will be counted by the server (captured by the predicate `v in votes`). With this property the server guarantees there cannot be any loss of valid votes.

4.7 Semantics of Application

We can now give the semantics of a web application. An application consists of cubes and services. The latter are components, so they are represented by their contracts. Formally, we represent an application by this structure:

$$App \;\stackrel{d}{=}\; (\mathsf{svc} :: \{Contract\},\ \mathsf{cube} :: \{Box\}) \tag{8}$$

If C is a set of boxes, let $[\!]C$ denote the parallel composition of all the boxes in C. Let A be an application. The Seuss semantics of A is the concrete program induced by A, which is just the parallel composition of all its services and cubes:

$$A.\mathsf{impl} \;\stackrel{d}{=}\; ([\!]c : c \in A.\mathsf{svc} : c.\mathsf{impl})\ [\!]\ ([\!]\ A.\mathsf{cube}) \tag{9}$$

Although this implementation is not visible, we can infer, from the cubes and the contracts, an abstract model for the application:

$$A.\mathsf{model} \;\stackrel{d}{=}\; ([\!]c : c \in A.\mathsf{svc} : c.\mathsf{smodel})\ [\!]\ ([\!]\ A.\mathsf{cube}) \tag{10}$$

$A.\mathsf{client}$ is A's worst allowed client. It is the one that tries all possible calls to the methods of A's cubes:

$$A.\mathsf{client} \;\stackrel{d}{=}\; ([\!]x : x \in A.\mathsf{cube} : c.\mathsf{smodel.env}) \tag{11}$$

Note that $A.\mathsf{client}$ is by definition an abstraction of any proper client of A.

Wrapping. Since semantically, $A.\mathsf{impl}[\!]client$ is a box, it can be treated as a component by providing a contract. Semantically, it becomes a service. In the implementation this may require some wrapping to make it SOAP-enabled. As a service it can be used to build larger applications.

5 Inference

Seuss provides a logic [14] for proving safety and progress properties. Although we have changed the definitions of Seuss temporal operators, it can be proven in a quite standard way that they maintain basic Seuss laws, e.g. using our general proof theory in [17]. We now add important results, namely theorems for inferring properties of an application from the contracts of its services —with just plain Seuss, this is not possible.

Let A be an application. Let $A.\mathsf{inv}$ denote the combined abstract invariant of A, which is the conjunction of the invariants specified by the contracts in A. Similarly, $A.\mathsf{concreteInv}$ denotes the combined concrete invariant of A. The latter cannot be inferred from the contracts. However, we just need to infer that properties inferred from A are consistent with it. Let *client* be a proper client of A (under $A.\mathsf{inv}$). We have:

Thm. 7 : Inferring Safety from Abstract Model

$$\frac{A.\mathsf{model}[\!]client,\ A.\mathsf{inv}\ \vdash\ p\ \mathsf{unless}\ q}{A.\mathsf{impl}[\!]client,\ A.\mathsf{concreteInv}\ \vdash\ p\ \mathsf{unless}\ q}$$

Proof: the Contract-Box relation (Def. 6) imposed on the services implies that $A.\mathsf{model}$ is a consistent abstraction of $A.\mathsf{impl}$:

$$A.\mathsf{concreteInv} \vdash A.\mathsf{model} \sqsubseteq A.\mathsf{impl} \tag{12}$$

It follows, by Thm. 5, that any unless property proven on the abstract model is also a property of the concrete system. □
For inferring progress we have:

Thm. 8 : Progress by Contract

$$\frac{c \in A.\mathsf{contract} \quad \wedge \quad p \mapsto q \in c.\mathsf{progress}}{A.\mathsf{impl}_{\triangleleft} [\![client,\ A.\mathsf{concreteInv} \vdash p \mapsto q}$$

Proof: by Def. 6, $c.\mathsf{progress}$ actually specifies this progress: $c.\mathsf{impl}_{\triangleleft} [\![c.\mathsf{env} \vdash p \mapsto q$. Imposing the constraint on the atomicity of method calls from Subsection 4.5, makes the rest of the application and the *client* act as a proper environment for c. Hence, by Thm. 4 the progress will be preserved in the entire system. □

Below is the dual of the theorem above, stating that progress solely driven by the *client*, assuming A's abstract model as the environment, will be preserved in the entire system:

Thm. 9 : Client Progress

$$\frac{client_{\triangleleft} [\![A.\mathsf{model},\ A.\mathsf{inv} \vdash p \mapsto q}{client_{\triangleleft} [\![A.\mathsf{impl},\ A.\mathsf{concreteInv} \vdash p \mapsto q}$$

Proof: follows from (12) and Thm. 4.

Example. Consider again the example we mentioned in the Introduction: we want a guarantee that a valid vote submitted to the `webVote` application in Figure 1 will eventually be counted. The property is promised by the `VoteServer` service in `webVote`. Now we can use Thm. 8 to conclude that the property will indeed be preserved in the system.

Consider also the property `info()` $\geq N$ unless false. It is an important safety property, stating that the application will not silently cancel an already counted vote. In order to verify its correctness, Thm. 7 says that we can do so against the *abstract* model of the application. This means isolated verification: we do not need the full code of the services!

We cannot infer everything from a contract, because it is just an abstraction. For example, the component `VoteServer` in Figure 2 will not silently insert a valid-but-fake vote. However, we cannot infer this from the contract in Figure 4.

6 Related Work

Formal methods have been used to specify and verify document related properties of web applications. Semantic Web [5] is currently popular as a framework

to define the semantics of documents, thus enabling reasoning over them, e.g. simply by using theorem provers. On top of it sophisticated properties can be specified e.g. as [12] that offers a query language, in the spirit of SQL, over documents. Automated verification has also been explored [22,9,1], though we will have to limit ourselves to simple document properties, e.g. the reachability of different parts of a web page from multiple concurrent frames. Web Cube logic focuses on temporal properties over the state maintained by a web application, rather than on document properties —these two aspects are complementary.

A Web Cube is primarily a programming model for constructing a web application. Although it is based on services composition, it is not a dedicated service orchestration language as e.g. BPEL, cl [8], or Orc [15]. Given a Web Cube application A, requests from a client are translated to calls to A's cubes' methods. In turn a method may perform a series of calls to multiple services, scripted as a plain Seuss statement. So, orchestration in Web Cube happens at the method level, and is consequently limited by the atomicity constraint over methods. Therefore, the full BPEL concurrency (of orchestration) cannot be mapped to Web Cube's orchestration. Though on the other hand we get a nice back box temporal logic for Web Cube, whereas this would be a problem for BPEL. Orc's [15] type of orchestration matches better to Web Cube. A top level Orc expression is atomic. So in principle it can be used to specify a cube's method. Formalisms like process algebra [6], Petri net [21], or event-based temporal logic [24] have been used to reason over service orchestration. These are more suitable for work-flow oriented style of orchestration (e.g. as in BPEL). In Web Cube calls to services may cause side effect on the services' persistent state. So, Web Cube uses a classical temporal logic which is more suitable to reason over temporal properties over persistent states.

Web Cube assumes a more classical development cycle, where Seuss is used to abstractly describe a web application. Properties are reasoned at this Seuss level. Actual implementation could be obtained by translating Seuss to an implementation language, e.g. Java. Language embedding [11] is also an interesting route to obtain implementation. Others have used refinement to develop an application [23]. Seuss is not a refinement calculus; refinement in Web Cube is used to bind contracts. In *reverse engineering* people attempt to do the opposite direction: to extract models from an existing implementation of a web application, e.g. as in [7,22,9,1]. The models are subsequently subjected to analysis, e.g. verification. Reverse engineering can yield high automation, but defining the procedure to extract the models is not trivial, especially if the programming model used at the model level differs too much from that of the implementation level. This is likely the case with Web Cube, since it tries to be high level, and hence hard to extract from e.g. an arbitrary Java code.

Compared to all the work mentioned above Web Cube is also different because of its component based approach.

References

1. M. Alpuente, D. Ballis, and M. Falaschi. A rewriting-based framework for web sites verification. In *Proc. 5th Int. W.shop on Rule-based Programming RULE*. Elsevier Science, 2004.
2. R.J. R. Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Also available as report A-1978-5.
3. R.J.R. Back and J. Von Wright. Refinement calculus, part II: Parallel and reactive programs. *Lecture Notes of Computer Science*, 430:67–93, 1989.
4. K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison-Wesley Publishing Company, Inc., 1988.
5. M.C. Daconta, L.J. Obrst, and K.T. Smith. *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management*. 2003.
6. A. Ferrara. Web services: a process algebra approach. In *Proceedings of 2nd International Conference Service-Oriented Computing (ICSOC)*, pages 242–251. ACM, 2004.
7. H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a tool for model-based verification of web service compositions and choreography. In *Proceeding of the 28th international conference on Software engineering*, pages 771–774. ACM Press, 2006.
8. S. Frolund and K. Govindarajan. cl: A language for formally defining web services interactions. Technical Report HPL-2003-208, Hewlett Packard Laboratories, 2003.
9. M. Haydar. Formal framework for automated analysis and verification of web-based applications. In *Proc. 19th IEEE Int. Conf. on Automated Software Engineering (ASE)*, pages 410–413. IEEE Computer Society, 2004.
10. R. Herk. Web functions, 2005. Master thesis, IICS, Utrecht Univ. No. INF/SCR-2005-014.
11. S. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering. In *Proc. 5th Int. Conf. on Functional Programming*, pages 280–292, 2000.
12. A.O. Mendelzon and T. Milo. Formal models of Web queries. In *Proc. of the 16th ACM Sym. on Principles of Database Systems (PODS)*, pages 134–143, 1997.
13. B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, 1992.
14. J. Misra. *A Discipline of Multiprogramming*. Springer-Verlag, 2001.
15. J. Misra. A programming model for the orchestration of web services. In *2nd Int. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 2–11, 2004.
16. I.S.W.B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Inst. of Information and Computer Sci., Utrecht University, 1995.
17. I.S.W.B Prasetya, T.E.J. Vos, A. Azurat, and S.D. Swierstra. !UNITY: A HOL theory of general UNITY. In *Emerging Trends Proceedings of 16th Int. Conf. Theorem Proving in Higher Order Logics*, pages 159–176, 2003.
18. 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, IICS, Utrecht Univ., 2003.
19. I.S.W.B Prasetya, T.E.J. Vos, A. Azurat, and S.D. Swierstra. A unity-based framework towards component based systems. In *Proc. of 8th Int. Conf. on Principles of Distributed Systems (OPODIS)*, 2004.
20. B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

21. K. Schmidt and C. Stahl. A petri net semantic for BPEL. In *Proc. of 11th Workshop Algorithms and Tools for Petri Nets*, 2004.
22. E. Di Sciascio, F.M. Donini, M. Mongiello, and G. Piscitelli. AnWeb: a system for automatic support to web application verification. In *SEKE '02*, pages 609–616. ACM Press, 2002.
23. G. Di Marzo Serugendo and N. Guelfi. Formal development of java based web parallel applications. In *Proc. of the Hawai Int. Conf. on System Sciences*, 1998.
24. M.P. Singh. Distributed enactment of multiagent workflows: temporal logic for web service composition. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 907–914. ACM Press, 2003.
25. C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
26. R.T. Udink. *Program Refinement in UNITY-like Environments.* PhD thesis, Inst. of Information and Computer Sci., Utrecht University, 1995.
27. T.E.J. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms.* PhD thesis, Inst. of Information and Computer Sci., Utrecht University, 2000.