A UNITY-based Framework towards Component Based Systems

I.S.W.B. Prasetya[*] T.E.J. Vos[†] A. Azurat[*] S.D. Swierstra[*]

Abstract

The paper offers a UNITY-based framework to model distributed applications which are built with a component based approach. The framework enables components to be abstractly specified in terms of contracts. Temporal properties are expressed and proven in the UNITY style. Compositional reasoning about components' properties, including progress, is supported. The semantical model is simple and intuitive.

[*]Informatica Instituut , Universiteit Utrecht , email: {wishnu,doaitse,ade}@cs.uu.nl.

[†]Instituto Tecnológico de Informática , Universidad Politécnica de Valencia , email: tanja@iti.upv.es.

Contents

1 Introduction

Consider an application built from smaller components. They interact by calling each other's operations. Some of the components may actually not reside in the application itself. Instead, they are owned and controlled by other applications, which may run on different machines —access to these external components may be facilitated by some kind of broker programs, though this may be transparent to the application itself. This is a quite typical component based model used in practice today, for example as adopted in COM, CORBA, and JavaBeans.

Because the components may run on separate machines, such an application is essentially a distributed system whose temporal properties have to be verified. For example, it will be essential to know that a third-party web-based voting service does not unauthorizedly alter the information in the incoming votes while the voting process is still going on. Verifying a property of a component based system is however more complicated because we may not have access to the source code of all its components. Instead, we have to rely on their specifications —also called contracts. To enable temporal properties to be inferred, the components will however have to offer a stronger kind of contracts. Merely specifying the pre- and post-conditions of an object's operations, for example as in OCL, is usually not sufficient.

UNITY is a formalism introduced by Chandy and Misra in 1988 to reason about distributed programs [5, 13]. It consists of a programming language to model programs, a specification language to express temporal properties, and a logic to prove them. The programming language is rather inconvenient for writing concrete programs, since it offers very little structuring constructs (UNITY has for example no `;` and no `while`). It does well however when used as a semantical model of concrete programs, or to describe abstract programs. Compared to LTL [12], UNITY's specification language is less expressive, though in exchange UNITY is simpler and more intuitive —and one can still express a lot of useful properties in UNITY.

This paper offers a UNITY-based framework to specify, using contracts, the components of a distributed application and to infer temporal properties of the application from the contracts of its components. UNITY is used in two ways: as a semantical model and as a specification language. The use of UNITY as a semantical model leads to a semantics which is simple and intuitive. To specify concrete components, we use the combination of UNITY programs and properties. This turns out to work quite well. Furthermore, we provide laws to allow global properties, including progress, of an application to be inferred in a compositional way from the properties of its components. Program refinement is used as a part of the component-contract relation. On purpose, the relation is made somewhat lenient, with the benefit that checking component-contract consistency is cheaper and that a component's author gets more flexibility in hiding aspects of his component from its contract (while still offering a consistent contract) —though the trade off is that the kind of progress properties that can be inferred from a contract is limited.

2 Overview on the Model

We will take the same basic model as in CORBA: an *application* consists of *clients* and *objects*. Both are computing entities, and to keep it simple, we assume that they run continuously. These computing entities interact by performing *operations* (methods) provided by the objects.

We define *object* as a computing entity which can only be interacted to by the environment through a set of operations provided by the object[1]. Furthermore, a developer developing an application cannot assume to have full information of the objects used to build the application. In particular, their source code may not be available. Clients are however assumed to be fully owned by the developer, and thus he has access to their source code.

We also say that objects are available as *components*. We define (software) component as a program that, for various reasons, only reveals limited amount of information about itself. We will require *commitment*. That is, a component is bound to realize whatever things it says about itself in the information it releases to its users[2]. Because of this binding nature, the revealed information is also called *contract*, and the object being bound by a contract is also called a *contractor*.

So, the only information about an object that an application developer can rely upon is the contract offered by the object.

We will use a hypothetical language, let us called it Dalang (*Distributed application development language*), for showing some examples in a more programming-like notation —hopefully it helps the reader to relate the framework to the familiar concepts from the practice. We will not pay much attention to the syntactical aspect of Dalang.

Figure 1 shows an example of a Dalang application. An application consists of objects and clients. The application in the example has two objects and one client. The objects are named `v` and `d`, the first is of the class `SimpleVotingSystem`, and the latter of the class `SimpleCalendar`. A `SimpleVotingSystem` object provides an operation allowing users to send votes; it checks the validity of the incoming votes; and it also provides an operation to count the votes. An example of a property which one may want to infer from such an object is that it will not silently add or remove votes. We cannot do this yet, since we have not seen `SimpleVotingSystem`'s contract —we will return to this later.

Figure 1 also shows a contract. It is the contract offered by the class `SimpleCalendar`. The `public` section of a contract shows the names and types of variables that the contractor object should publicly expose. So, `SimpleCalendar` only offers one public variable called `current`, intended to hold the current date. The `operation` section lists available operations. In the

[1]This is consistent with Szyperski's definition of *object* (essentially: an object is something that has state, behavior, and encapsulation) [18], which is quite commonly accepted.

[2]This is also consistent with Szyperski's definition of *component* [18], essentially: component is a unit of composition with contractually specified interfaces and subject to composition by third parties. Our definition is stricter by saying the only knowledge we can rely on, placing ourselves as a third party, about a component is its contracts.

```
application votingService
   public
     v :: SimpleVotingSystem ;
     d :: SimpleCalendar

   client superviseVoting(v,d)
     closingDate :: Date = 01/01/2004 ;
     today       :: Date = 00/00/0000
     do
        d.getDate(today)
     [] today>=closingDate --> v.count()

contract SimpleCalendar
  public
    current :: Date
  operation
    getDate(&today::Date) = do today:=current
  safety model
    current := current + unittime
  invariant
    current>=0
  progress
    !D. true |--> current>=D
```

Figure 1: *The figure shows a simple application to do electronic voting. It uses two objects: one for enabling users to send votes, and the other one is a simple calendar system to keep track of the current date. The figure also shows the contract of the calendar object.*

`SimpleCalendar` example only one operation is offered, namely `getDate`, which returns the value of `current`. The contract does not tell us how exactly a `SimpleCalendar` object maintains the current date, though the `safety model` may provide a partial answer. It shows a UNITY program serving as an abstraction of the actual object. In the example, it says that each step of a `SimpleCalendar` object either increases `current` or leaves it unchanged —the latter possibility is implicitly imposed by the object-contract relationship; we will return to this later. It is not a strong model. Nevertheless, one can infer from it that `current` will not decrease and thus the date maintained by a `SimpleCalendar` object will never move backwards.

The `invariant` section specifies a property which is always maintained through out the object's life. In this case it says that `current` is never negative.

The `progress` section specifies progress properties promised by the object. In the example it says that `current` will increase, and thus the date maintained by a `SimpleCalendar` object *will* move forward.

The client in the example regularly checks the date. If the closing date is passed, it will call the `count` operation of `v`, with the effect that all incoming valid votes up to that point will be counted.

As we use objects to build an application, it is important that an object will always behave as its contracts specifies, regardless how the object is being used in the application. In other words, we want the invariant and all progress properties specified in the contract to hold, not only when the object is executed alone, but also when it is composed in any (proper) environment. Since parallel composition tends to destroy temporal properties, we will introduce new set of extended UNITY operators describing more robust classes of temporal properties. We also have to define an abstraction relation between programs, so that safety properties inferred from the safety model in the contract, which is an abstraction of the actual contractor object, will be preserved by the object itself.

We will not venture into complex features, such as inheritance and the ability to pass object reference, or to pass an entire object, through an operation call. Furthermore, our model is an abstract model: details of implementational nature, such as parameters marshaling, object deployment, and optimization of resources' utilization will not be visible in the model.

Notational Convention

Tuples are used here to represent composite structures, such as an object. We will use tuples with selector functions. For example,

$$Object = (\mathsf{prg} :: Program, \mathsf{ops} :: \{Operation\})$$

define a type $Object$ consisting of two-elements tuples. If $x = (P, M)$ is a value of this type, then $x.\mathsf{prg} = P$ and $x.\mathsf{ops} = M$.

The following lists the default use of letters to denote various concepts:

1. program variables: u, v, w
2. set of variables: U, V, W
3. predicate: p, q, r, s
4. invariant: i, j
5. temporal property, such as $p \mapsto q$: Φ
6. action: a, b, c
7. set of action: Σ
8. program: P, Q, R
9. abstract program: A, B, C
10. abstract environment of a program: B
11. concrete environment of a program: Q

12. object: x, y, z
13. classes: X, Y, Z
14. set of objects: χ, ψ
15. operation: k, l, m
16. set of operations: K, L, M
17. contract: c, d
18. application: $\mathcal{A}, \mathcal{B}$
19. vote (in the running example): v

3 Basic

Predicate Confinement

Predicates are used to 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. For example, $x > y+z$ constrains the values of x, y and z, but does not constrain the values other variables. So, $x > y + z$ is *confined* by $\{x, y, z\}$. Confinement enables us to specify how sensitive a predicate is to interference. We write p, q **conf** V to abbreviate p **conf** V and q **conf** V. As a rule of thumb, if V is the set of free (non-logical) variables of an expression e, then e is confined by V.

Action

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 $\mathcal{P}(\mathsf{State})$. We use the following syntax to describe *basic* actions:

$$\begin{array}{lcl} \langle \textit{basic action} \rangle & ::= & \texttt{skip}\ \langle \textit{set of variables} \rangle ? \\ & | & \langle \textit{assignment} \rangle \\ & | & \langle \textit{expr} \rangle\ \texttt{-->}\ \langle \textit{assignment} \rangle \end{array}$$

If V is a set of variables, $\mathsf{skip}\ V$ is an action that leaves the variables in V unchanged, but can do anything to the rest. An assignment can also simultaneously assign to multiple variables. The meaning of g `-->` a is that a will be executed if g is true, otherwise the action behaves as skip.

Furthermore, if a and b are actions, $a \sqcup b$ is an action that either does 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 $(\sqcup a : a \in \Sigma : a)$. $\mathsf{skip}\ V$ is an action that leaves variables listed in V unchanged, but can do anything to the other variables.

If a is an action, the syntax $\{\texttt{var}\ x;\ a\}$ is used to introduce a local variable x. The meaning is expressed in terms of Hoare triple as follows:

$$\{p\}\ \{\texttt{var}\ x;\ a\}\ \{q\}\ \overset{d}{=}\ \{p\}\ a[x'/x]\ \{q\}$$

where $a[x'/x]$ means the action obtained by replacing x in a with a fresh variable x'.

Sometimes it is convenient to borrow the syntax typically used to describe non-atomic state transition, usually called *statement*, to describe an action. Let us assume the following simple syntax to build statements:

$$\begin{array}{lcl} \langle \textit{statement} \rangle & ::= & \langle \textit{assignment} \rangle \\ & | & \langle \textit{statement} \rangle \ \texttt{;} \ \langle \textit{statement} \rangle \\ & | & \texttt{if} \ \langle \textit{expr} \rangle \ \texttt{then} \ \langle \textit{statement} \rangle \ (\texttt{else} \ \langle \textit{statement} \rangle)? \\ & | & \texttt{while} \ \langle \textit{expr} \rangle \ \texttt{do} \ \langle \textit{statement} \rangle \end{array}$$

Their meaning is as usual. For any terminating statement S, we use the notation `action` S to convert S into an action.

Action Refinement

We define the following notion of refinement over actions —it is a variant of the standard one, e.g. as in [3]. Let V be a set of variables, and i is a predicate, intended to be an invariant. An action b refines another action a with respect to V and i, if b can simulate whatever a can do on the variables in V, assuming i holds initially. Formally:

$$V, i \vdash a \leq b \ \ \overset{d}{=} \ \ (\forall p, q : p, q \ \mathbf{conf} \ V : \{i \wedge p\} \ a \ \{q\} \ \Rightarrow \ \{i \wedge p\} \ b \ \{q\})$$

The reverse of refinement is also called *abstraction*. So, if b refines a we can also say that a is an abstraction of b.

An action b is said to *weakly refine* a, with respect to V, if it either refines a, or it behave as $\mathsf{skip}\ V$. Formally:

$$V, i \vdash a \sqsubseteq b \ \ \overset{d}{=} \ \ V, i \vdash (a \sqcup \mathsf{skip}\ V) \leq b$$

4 UNITY

We will use the original UNITY operators from [5] as a base for our extension. We could have used those of new-UNITY [13], since both set of operators are in principle of equal strength. However, the choice is a subjective one, to us the older operators are simply more intuitive.

4.1 Programs

We will represent a UNITY program P by a tuple of this type:

$$\textit{Prog}_{\text{UNITY}} \ \overset{d}{=} \ (\mathsf{acts} :: \{\textit{Action}\}, \mathsf{init} :: \textit{Pred}, \mathsf{pub} :: \{\textit{Var}\}, \mathsf{pri} :: \{\textit{Var}\})$$

$P.\mathsf{init}$ is a predicate specifying P's possible initial states, $P.\mathsf{pub}$ is the set of public (shared) variables offered by P, and $P.\mathsf{pri}$ is the set of P's private (local) variables. The notation $P.\mathsf{var}$ is used 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 every action $a \in P.\mathsf{acts}$ is non-abortive, which means that for every state s, $a\ s$ is non-empty.

A UNITY program runs forever. Its actions is executed interleavingly, where at each step an action is selected non-deterministically. Each execution of the program is assumed to be *fair* in the sense that each action of the program is executed infinitely many often. UNITY actions may be guarded, though, unlike in LTL with strong fairness, when an action is selected to be executed, and its guard evaluates to false, then it will simply be executed as a skip.

We do not expect real programs to be written in UNITY, but we will assume that they can be translated to UNITY programs —see for example [12][3]. So, in the rest of the paper, we will simply represent programs by their UNITY translations.

We define a UNITY program to be an *abstract program* if it has no local variables, otherwise it is a *concrete program.* An application is built by composing concrete programs, whose abstract behavior will be captured by an abstract program. Note that the role of an abstraction is primarily to provide partial information about the thing it abstracts from. So, an abstraction does not have to be complete.

When composing *different* programs, we assume that each component program is given a unique name space to name its private variables. For example, this can be achieved by prefixing the names of all private variables of a program with the program's name. We will however not concern ourselves any further with the used naming scheme —the unique name space assumption is sufficient for us.

As a consequence, when composing, for example, two different programs P and Q, we can assume that the names in $P.\mathsf{pri}$ and $Q.\mathsf{pri}$ do not clash with the names in, respectively, $Q.\mathsf{var}$ and $P.\mathsf{var}$. Composing two programs means letting them to run in parallel. The behavior of the parallel composition of P and Q is modelled by the program $P \| Q$ defined as follows:

Definition 4.1 : Parallel Composition

$$P \| Q \overset{d}{=} (P.\mathsf{acts} \cup Q.\mathsf{acts},\ P.\mathsf{init} \wedge Q.\mathsf{init},\ P.\mathsf{pub} \cup Q.\mathsf{pub},\ P.\mathsf{pri} \cup Q.\mathsf{pri})$$

□

4.2 Program Refinement

We will use the following simple notion of refinement on UNITY programs. The definition mentions the notion 'invariant'. It is a predicate which holds throughout the program's executions —its formal definition will be given later.

Definition 4.2 : Program Refinement and Abstraction
Let V be a set of variables and i be a predicate, intended to be a strong invariant of P. We define:

[3]It describes the translation from a simple programming language to fair transition systems, which has a similar structure as UNITY programs

1. $V, i \mid\!\!-\!- P \sqsubseteq Q$
   $\overset{d}{=}$
   $P.\mathsf{pub} \subseteq Q.\mathsf{pub} \ \wedge\ P.\mathsf{pri} \subseteq Q.\mathsf{pri} \ \wedge\ Q.\mathsf{init} \Rightarrow P.\mathsf{init}$
   $\wedge$
   $(\forall b : b \in Q.\mathsf{acts} : V, i \ \vdash\ \sqcup\, P.\mathsf{acts} \ \sqsubseteq\ b)$

2. $i \vdash P \sqsubseteq Q \ \overset{d}{=}\ \ P.\mathsf{var}, i \mid\!\!-\!- P \sqsubseteq Q$

□

So, under the invariance of i, $V, i \vdash P \sqsubseteq Q$ means that every action of Q behaves, with respect to the variables in V, no worse than some action of P, or it just skips V (note the use of weak refinement at the action level). When this is the case we say that Q is a refinement of P, and that P is an *abstraction* of Q.

4.3 Properties

In the following, we will introduce several new sets of extended UNITY operators that will be especially useful to describe progress properties of an object, under any proper environment, in its contract. The extensions are in the style of [15] and are needed since the progress properties specified within a contract are only of use when it can be easily inferred that they will be preserved regardless of the environment the contractor is composed with.

Let us first give a definition of invariant, since we will use it later. A predicate i is called a *strong invariant*[4] of P, denoted by $P \vdash \mathsf{sinv}\ i$, if it holds initially, and it is maintained by every action of P. So:

Definition 4.3 : Invariant

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

□

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

The first set of new operators defined below specify additional assumptions on a given property Φ of a program P so that we can infer what kind of environment of P can preserve the property Φ. They are defined as follows:

Definition 4.4 : Extended UNITY operators (I)

1. $P, i \ \mid\!\!-\!-\ \ p\ \mathsf{unless}\ q\ \mathsf{using}\ V$
   $\overset{d}{=}$
   $P\ \vdash\ \mathsf{sinv}\ i\ \ \wedge\ \ p, q\ \mathbf{conf}\ V\ \ \wedge\ \ (\forall a : a \in P.\mathsf{acts} : \{i \wedge p \wedge \neg q\}\ a\ \{p \vee q\})$

[4]We are going to use invariants to parameterize UNITY properties, in the style of Sanders [16]. Strong invariants are however used here instead of just invariants (predicates that hold through out any execution of a given program) as in [16], because the later cause a certain technical problem [14].

2. $P, i\ \ |\!\!-\!\!-\ \ p\ \textsf{ensures}\ q\ \textsf{using}\ V$
$\overset{d}{=}$
$P, i\ \ |\!\!-\!\!-\ \ p\ \textsf{unless}\ q\ \textsf{using}\ V\ \wedge\ (\exists a : a \in P.\textsf{acts} : \{i \wedge p \wedge \neg q\}\ a\ \{q\})$

$\Box$

In [15] the V parameter can be used to specify interference at the run-time, which is a dynamic property. In the framework here V is fixed to $P.\textsf{var}$. Consequently, it serves more as a static constraint. So, what we will do now is to hide this parameter. However, for the purpose of proving some of the main theorems it will be necessary to expose the V again.

We define now an analogous set of operators, in which V is fixed to $P.\textsf{var}$:

Definition 4.5 : Extended UNITY Operators (II)
For any property Φ of the form p unless q, p ensures q we define:

$$P, i\ \vdash\ \Phi\ \ \overset{d}{=}\ \ P, i\ \ |\!\!-\!\!-\ \ \Phi\ \textsf{using}\ P.\textsf{var}$$

$\Box$

Our previously defined refinement relation preserves unless properties:

Theorem 4.6 : Safety Preservation by Refinement

$$\frac{\begin{array}{c} P, i\ |\!\!-\!\!-\ p\ \textsf{unless}\ q\ \textsf{using}\ V \\ V, j\ |\!\!-\!\!-\ P \sqsubseteq Q\ \ ,\ \ Q \vdash \textsf{sinv}\ j\ \ ,\ \ j \Rightarrow i \end{array}}{Q, j\ |\!\!-\!\!-\ p\ \textsf{unless}\ q\ \textsf{using}\ V}$$

$\Box$

proof:

$P, i\ |\!\!-\!\!-\ p\ \textsf{unless}\ q\ \textsf{using}\ V$

$\Rightarrow$ { Definition 4.4 of unless }

$(\forall a : a \in P.\textsf{acts} : \{i \wedge p \wedge \neg q\}\ a\ \{p \vee q\}$

$\Rightarrow$ { definition of $\sqcup$ }

$\{i \wedge p \wedge \neg q\}\ \sqcup P.\textsf{acts}\ \{p \vee q\}$

$=$ { p **conf** V, so skip V maintains p }

$\{i \wedge p \wedge \neg q\}\ (\sqcup P.\textsf{acts}) \sqcup \textsf{skip}\ V\ \{p \vee q\}$

$\Rightarrow$ { $j \Rightarrow i$, pre-condition strengthening }

$\{j \wedge p \wedge \neg q\}\ (\sqcup P.\textsf{acts}) \sqcup \textsf{skip}\ V\ \{p \vee q\}$

$\Rightarrow$ { $V, j \vdash P \sqsubseteq Q$, so for all $b \in Q.\textsf{acts}$: $V, j \vdash (\sqcup P.\textsf{acts}) \sqcup \textsf{skip}\ V \leq b$ }

$(\forall b : b \in Q.\textsf{acts} : \{j \wedge p \wedge \neg q\}\ b\ \{p \vee q\}$

$=\qquad \{\ j$ is a strong invariant of $Q\ \}$

$Q, j \mid\!\!-\!\!- p\ \mathsf{unless}\ q\ \mathsf{using}\ V$

end

It follows, that if A is an abstractions of a program P, any unless property of $A \| Q$ is also a property of $P \| Q$. More precisely:

Theorem 4.7 : Safety Preservation
Let A, P, and Q be such that $A.\mathsf{var} \cap Q.\mathsf{var} = P.\mathsf{var} \cap Q.\mathsf{var}$. We have:

$$\frac{\begin{array}{c} A\|Q, i \ \vdash\ p\ \mathsf{unless}\ q \\ j \vdash A \sqsubseteq P \quad , \quad P\|Q \vdash \mathsf{sinv}\ j \quad , \quad j \Rightarrow i \end{array}}{P\|Q, i \ \vdash\ p\ \mathsf{unless}\ q}$$

□

proof: First notice that $A.\mathsf{var} \cap Q.\mathsf{var} = P.\mathsf{var} \cap Q.\mathsf{var}$ implies that $(Q.\mathsf{var} - A.\mathsf{var}) \cap A.\mathsf{var} = (Q.\mathsf{var} - A.\mathsf{var}) \cap P.\mathsf{var} = \emptyset$. So, by Theorem A.8, the refinement in the premise implies this:

$$A.\mathsf{var} \cup (Q.\mathsf{var} - A.\mathsf{var}),\ j\ \vdash\ A \sqsubseteq P$$

which is equal to:

(a) $\quad (A\|Q).\mathsf{var},\ j\ \vdash\ A \sqsubseteq P$

We derive now:

$A\|Q, i \vdash p\ \mathsf{unless}\ q$

$=\qquad \{$ Definition 4.5 $\}$

$A\|Q, i \vdash p\ \mathsf{unless}\ q\ \mathsf{using}\ (A\|Q).\mathsf{var}$

$\Rightarrow\qquad \{$ by Theorem A.9 (a) implies $(A\|Q).\mathsf{var}, j \vdash A\|Q \sqsubseteq P\|Q$;
then use Theorem 4.6 $\}$

$P\|Q, j \vdash p\ \mathsf{unless}\ q\ \mathsf{using}\ (A\|Q).\mathsf{var}$

$\Rightarrow\qquad \{\ A \sqsubseteq P$ implies $A.\mathsf{var} \subseteq P.\mathsf{var}$, so $(A\|Q).\mathsf{var} \subseteq (P\|Q).\mathsf{var}$;
then use Theorem A.11 $\}$

$P\|Q, j \vdash p\ \mathsf{unless}\ q\ \mathsf{using}\ (P\|Q).\mathsf{var}$

$=\qquad \{$ Definition 4.5 $\}$

$P\|Q, j \vdash p\ \mathsf{unless}\ q$

end

Abstraction will be used later as part of the contract of an object. Recall however, that with our definition of $\sqsubseteq$ we can abstract a program P by replacing some its actions with skip. This is a powerful form of abstraction, but it also means that the relation will *not* in general preserve progress properties. This is a deliberate choice. Preserving progress properties across refinement typically requires various internal details of P to be carried over to its abstraction, thus requiring a weaker notion of abstraction. We favour a stronger abstraction relation (more lenient refinement relation) as it gives users more choices in deliberately hiding implementational details. The more we hide, the less we can prove of course, but on the other hand, verification becomes quite often easier.

In order to be able to reason about the preservation of progress properties, we will require that the author of a contract c specifies a set of progress properties which can be preserved if c's contractor P is composed with some environment E. Consider the following property:

$$P \| E, i \vdash p \mapsto q$$

Suppose we know that this progress is driven solely by P. If E 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 will however have to introduce a new set of extended UNITY operators, where the notion of "progress Φ is driven solely by P" as above can be specified. This is captured by the following:

Definition 4.8 : Extended UNITY operators (III)
Let P and B be UNITY programs. We define:

1. $P_{\lhd} \| B,\ i\ \vdash\ p \ \mathsf{ensures}\ q$
   $\overset{d}{=}$
   $P \| B,\ i\ \vdash\ p\ \mathsf{unless}\ q\ \wedge\ (\exists a : a \in P.\mathsf{acts} : \{i \wedge p \wedge \neg q\}\ a\ \{q\})$
2. $P_{\lhd} \| Q,\ i \vdash p \mapsto q$ is defined such that $(\lambda p, q.\ P_{\lhd} \| Q,\ i \vdash p \mapsto q)$ is the smallest transitive and disjunctive closure of $(\lambda p, q.\ P_{\lhd} \| Q,\ i \vdash p\ \mathsf{ensures}\ q)$.

□

Below is the theorem that makes it possible to specify progress properties in a contract in such a way that their preservation can be inferred regardless of the environment its contractor is composed with. More specifically, the theorem states that every progress property from p to q made by P, when specified in terms of $P_{\lhd} \| E$, will be preserved when P is composed with any program Q that refines E.

Theorem 4.9 : Preservation of $\mapsto$

$$\frac{\begin{array}{c} P_{\lhd} \| B, i \vdash p \mapsto q \\ j \vdash B \sqsubseteq Q \quad , \quad P \| Q \vdash \mathsf{sinv}\ j \quad , \quad j \Rightarrow i \end{array}}{P \| Q, i \vdash p \mapsto q}$$

□

proof: We will prove the theorem using $\mapsto$ induction —see for example [5, 13] for explanation on how it works. The transitivity and disjunctivity cases are trivial. So, it suffices to show the $\mathsf{ensures}$ case:

$$
\begin{array}{ll}
 & P_{\lhd}\|B, i \vdash p \ \mathsf{ensures}\ q \\
= & \{\ \text{Definition 4.8}\ \} \\
 & P\|B,\ i\ \vdash\ p\ \mathsf{unless}\ q\ \wedge\ (\exists a : a \in P.\mathsf{acts} : \{i \wedge p \wedge \neg q\}\ a\ \{q\}) \\
\Rightarrow & \{\ \text{Theorem 4.7}\ \} \\
 & P\|Q,\ j\ \vdash\ p\ \mathsf{unless}\ q\ \wedge\ (\exists a : a \in P.\mathsf{acts} : \{i \wedge p \wedge \neg q\}\ a\ \{q\}) \\
\Rightarrow & \{\ j \Rightarrow i, \text{ pre-condition strengthening}\ \} \\
 & P\|Q,\ j\ \vdash\ p\ \mathsf{unless}\ q\ \wedge\ (\exists a : a \in P.\mathsf{acts} : \{j \wedge p \wedge \neg q\}\ a\ \{q\}) \\
= & \{\ \text{Definition 4.8}\ \} \\
 & P_{\lhd}\|Q, j \vdash p\ \mathsf{ensures}\ q \\
\Rightarrow & \{\ \mapsto \text{ is a closure of } \mathsf{ensures}\ \} \\
 & P_{\lhd}\|Q, j \vdash p \mapsto q
\end{array}
$$

end

5 Object and Operation

In our model, an *object* is a program that exposes some of its variables to its environment. For simplicity we assume that such a program runs forever. This program is called the *internal* program of the object. The exposed variables are called *public* or shared variables. Access to these variables are however restricted: the environment can only inspect or update them via a set of operations provided by the object. Any program can be deployed as an object by encapsulating it with the necessary interface implementing the above restriction for accessing public variables. We will however not concern ourselves with the detail of such an interface.

Figure 5.1 shows a Dalang class of objects called `Simplevotingsystem`. Each object of this class will have the specified set of variables and operations, and `main`. The latter describes the internal program of the object. In principle `main` can be written in any language. We assume that it can be translated to UNITY. For simplicity, in the example `main` is written directly in UNITY.

In the example, a `Simplevotingsystem` object offers three operations. The operation `vote` allows users to send votes to the objects. The operation `count` causes the object to count collected votes. Only 'valid' votes will be counted. The result of the counting is reflected in the variable `accept`. It is set to `True` if the object can count at least 100 valid votes, and else `False`. Calling `count` also closes the voting process: after this incoming votes will be ignored.

The operation `result` allows a user to inspect the result of votes counting. The result is passed in the return parameter `r`. It is set to `[]` if the voting process is not closed yet (`count` has not been called). Else it will return the value of `[accept]`.

The `main` section describes the object's internal program. In the example it is written in UNITY. What the program does is to keep moving the first element of `votes` to an internal buffer `h`. It will subsequently check if the vote in `h` is a new and a valid vote. If so it will be moved to `validvotes`, else it will be deleted from `h`.

Note that the program allows votes validation to be executed in parallel with the buffering of the incoming votes. Moreover, it takes care that duplicate votes will not be counted.

5.1 Semantical Model

We will semantically model an object x with a tuple of this type:

$$Object = (\mathsf{prg} :: Prog_{\mathrm{UNITY}}, \mathsf{ops} :: \{Operation\})$$

where $x.\mathsf{prg}$ models the object's internal program, and $x.\mathsf{ops}$ is the set of operations offered by the object. *Operation* denote the universe of operation —its structure will be detailed latter. In the example in Figure 5.1, the the `main`, `public`, and `private` sections constitute the prg part; and the `operation` section constitutes the ops part.

The code in Figure 5.1 specifies however a class of objects. If X is a class, we write $x :: X$ to mean that x is an object obtained by renaming every variable v in the code of X with $x.v$ —we assume that this does not cause any name collision. We also say then that x is an object or an instance of class X.

We will also overload the selectors used on programs so that they also work on objects, with their (the selectors') meaning unchanged. So for example, $P.\mathsf{pub}$ denotes the the set of all public variables of the program P. With the same notation, if x is an object, $x.\mathsf{pub}$ denotes the set of all x's public variables, which is just equal to $x.\mathsf{prg}.\mathsf{pub}$.

Furthermore, we also overload $\|$ so that $x\|Q$ means $x.\mathsf{prg}\|Q$, and similarly for $P\|y$. Similarly, we also overload ${}_\triangleleft\|$.

Two objects are said to have an *identical interface* if they offer the same set of public variables and operations.

An operation will be described by a structure of this form:

$$mname(y_1, \ldots, y_k) = \ \mathsf{do}\ \ b$$

where *mname* is the name of the operation, followed by a list of its formal parameters, and b is an *action* describing the operation's body. The names of the formal parameters are assumed not to collide with the names of any variable of x. Parameters are passed either by value or by reference, but passing aliases

```
class SimpleVotingSystem

  public
   votes      :: [Int] := []    ;
   validvotes :: [Int] := [] ;
   accept  :: Bool  := False ;
   isOpen  :: Bool  := True

  private
    h :: [Int]           := [] ;
    checkingDone :: Bool := False

  operation
    vote (key::Int) = do if isOpen then votes := key:votes

    count() = do {isOpen := False ;
                  accept := length validvotes > 100 }

    result(&r::[Bool]) = do if isOpen then r:=[] else r:=[accept]

  main
     ~null votes /\ null h --> h:=[head votes]
  [] {var k::Int;
      atomic{if ~null h
             then { k := head h ;
                    if valid k /\ ~(k in validvotes)
                       then validvotes := k:validvotes ;
                    checkingDone := True }}}
  [] checkingDone
     --> h,votes,checkingDone := [], tail votes, False
```

Figure 2: *The figure shows a class called* `SimpleVotingSystem`*. Each object of this class will have the specified variables, operations, and* `main`*. The latter specifies the object's internal program. In the example it is described in UNITY.*

by reference is *not* allowed[5]. Furthermore, an operation has no access to global information, other than the public variables of the object it belongs to.

The execution of an operation is assumed to be *atomic*. This may not be efficient, for example if it works on a large data structure. There are ways to infer which operations will actually not interfere with each others, and hence can be safely executed in parallel. We will however abstract away this issue: it is up to the implementator of the underlying computation model to optimize the utilization of the objects.

In the sequel, to keep it simple, when discussing formal aspects of operations we will assume that their headers have the following simple form:

$$mname(y, r)$$

where y is a passed-by-value parameter used to pass input data to $mname$, and r is a passed-by-reference parameter used to hold the return value of $mname$. If x is an object to which $mname$ belongs to, then a call to $mname$ is denoted by $x.mname(e, z)$. If b is the body of $mname$, the effect of such a call is equivalent to:

$$\texttt{atomic}\{\texttt{var}\ y, r;\ y := e;\ b;\ z := r\}$$

5.1.1 Object Properties

Since the environment of an object x can only interact with it through its operations, the worst possible environment of x can be characterized by the following UNITY program:

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

where Σ is a set of actions modelling all possible calls to the operations in $x.\mathsf{ops}$:

$$\Sigma \ =\ \{\texttt{var}\ y, r;\ x.m_1(y, r)\} \ \| \ \ldots \ \| \ \{\texttt{var}\ y, r;\ x.m_k(y, r)\}$$

for all operations $m_1, \ldots, m_k \in x.\mathsf{ops}$.

So, any real or *proper* environment Q of x is a refinement of $x.\mathsf{env}$. Formally:

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

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

As an example, consider an object `x::Simplevotingsystem` —see again Figure 5.1. Note that the only action that can add a new element to `votes` is the operation `vote`, which can only do so if the `isOpen` flag is true. It follows that we have this property:

$$\begin{array}{l} \mathsf{x} \| \mathsf{x.env},\ \mathsf{true} \\ \vdash \\ \neg \texttt{x.isOpen} \wedge v \notin \texttt{x.votes}\ \ \mathsf{unless}\ \ \mathsf{false} \end{array} \qquad (1)$$

[5] Passing aliases by reference in the distributed setting will explode the reasoning complexity, if it is not already complicated enough, even without concurrency. So, we prefer to exclude them.

for any value v. It says that if the voting is already closed, no new votes can be added. Note that the property is an unless property over $x \| x.\mathsf{env}$: it will be preserved in the composition of x with any proper environment of x.

As another example, one can show that x satisfies the following progress property, saying that every valid vote in `votes` will eventually be copied to `validvotes`:

$$\begin{array}{l} \mathtt{x}_{\triangleleft} \| \mathtt{x.env},\ \mathtt{i} \\ \vdash \\ v \in \mathtt{filter\ valid\ x.votes} \ \mapsto\ v \in \mathtt{x.validvotes} \end{array} \tag{2}$$

for any value v. The property is a $\mapsto$ property over $x_{\triangleleft} \| x.\mathsf{env}$, which will be preserved in the composition of x with any proper environment of x. The `i` above is some invariant of x and $x.\mathsf{env}$. Note that in this example we will need a sufficiently strong `i` to be able to infer the progress specified above. For example, true will not be sufficient. For completeness, we include a possible `i` below.

$$\begin{array}{l} \neg\mathtt{null\ x.h} \ \Rightarrow\ \neg\mathtt{null\ x.votes} \wedge (\mathtt{head\ x.h} = \mathtt{head\ x.votes}) \\ \wedge \\ \mathtt{x.checkingDone} \ \Rightarrow\ \neg\mathtt{null\ x.h} \\ \wedge \\ \mathtt{x.checkingDone} \wedge \mathtt{valid(head\ x.h)} \ \Rightarrow\ (\mathtt{head\ x.h} = \mathtt{head\ x.validvotes})) \\ \wedge \\ (\forall i, j : i, j \in \mathtt{validvotes} : i = j) \wedge (\forall i : i \in \mathtt{validvotes} : \mathtt{valid}\ i) \end{array} \tag{3}$$

Abstraction Relation on Objects

We can lift the abstraction (refinement) relation between programs to the object level. We will only define the relation between two objects with identical interface[6].

Definition 5.1 : Abstracting Object
Let x' and x be two objects with identical interface. We define:

$$i \vdash x' \sqsubseteq x \ \stackrel{d}{=}\ i \ \vdash\ x'.\mathsf{prg} \sqsubseteq x.\mathsf{prg}$$

□

Note that, just as with the refinement relation on programs, the above relation only preserves safety. So, any property of the form p unless q proven on an abstraction x' of an object x, will also be a property of x.

[6]In a more general setting, one may want to allow operations to be refined as well. We will not do so here.

```
contract SimpleVotingSystem
  safety model
       ~ null votes  --> votes := tail votes
    [] {var k::Int;
        atomic{ if ~ null votes
                then { k := head votes ;
                       if   valid k /\ ~(k in valid votes)
                       then validvotes := k : validvotes }}}

  invariant
    (!i,j. i,j in validvotes ==> (i=j)) /\
    (!i.   i in validvotes ==> valid i)

  progress
    !v. v in filter valid votes |--->  v in validvotes
```

Figure 3: *The figure shows the contract offered by the class* `SimpleVotingSystem` *from Figure 5.1. The* `public` *and* `operation` *sections are here omitted to mean that they are identical to that of the class.*

6 Contract

An object does not release full information about itself to its environment. Instead, the object's users have to rely on a so-called *contract* offered by the object to figure out what the object does. A contract is binding, as the object is obliged to realize anything it commits in the contract. A simple form of contract simply lists the header of the operations offered by the object. A contract can be strengthened by putting more information in it, thus enabling the users to infer more properties about the object. Of course, strengthening a contract also makes an object less reusable, and its verification more expensive (it is then up to the object's producer to decide where to draw the line). Just keep in mind that the main purpose of a contract is to specify *some* properties of an object —it does not have to capture all of its properties. We will restrict ourselves here to the scenario where every object is associated to only one contract.

An abstraction relation, as formally defined in the previous section, can serve well as the base of a contract. We can use an abstract object a as a contract of a concrete object x. The definition of $\sqsubseteq$ tells us exactly how to check if x will indeed fulfil a. We use a quite powerful abstraction relation, which gives considerable freedom to developers in deciding how detailed a is (how much aspects of x we want to expose in a). This however has to be traded with something else: the preservation nature of the relation becomes weaker. As mentioned before, only safety properties proven on an abstract model a can be guaranteed to be inherited by the corresponding concrete object x. Progress properties of x cannot in general be inferred from a. So, if the producer of x wants to expose a progress property, he will have to include the property in the contract.

Figure 6 shows a Dalang example of a contract. The contract belongs to the class `SimpleVotingSystem` from Figure 5.1. The `safety model` section describes an abstraction, in the form of a UNITY program, of objects of the class `SimpleVotingSystem`; and the `progress` section specifies progress properties guaranteed by those objects. Implicitly here, the contract of a class C also exposes the `public` and `operation` sections of C[7].

As an example of how properties of an object can be inferred from its contract, consider an object `x::SimpleVotingSystem`. We can infer from the safety model in the contract of `SimpleVotingSystem` that no element from `validvotes` will ever be deleted, nor can an arbitrary value be added to it. The only to add a new value to `validvotes` is by inserting `head votes`. Stronger yet: the property is also maintained by $x.\mathsf{env}$. Formally:

$$\begin{array}{l} \mathsf{a}\|\mathsf{x.env}, \mathsf{true} \\ \vdash \\ (\mathtt{x.validvotes} = \sigma) \ \mathsf{unless} \ (\mathtt{x.validvotes} = \mathtt{head}\ \mathtt{x.votes} : \sigma) \end{array} \tag{4}$$

for all value σ. Here, a is the contract's safety model. Since $a \sqsubseteq x$, it follows then that the above property will also be a property of $x\|x.\mathsf{env}$.

Notice that the safety model in the example hides the fact that an actual `SimpleVotingSystem` object uses private variables `h` and `checkingDone` to synchronize parallel execution of votes buffering and votes validation. Because of this abstraction, we will not be able, for example, to infer from the safety model that the object will not arbitrarily delete a valid vote from its memory: an actual `SimpleVotingSystem` object regularly moves the head element of `votes` to the internal buffer `h`; however, since `h` is not visible in the contract, this action will be seen as a complete deletion of a vote.

The code in in Figure 6 actually defines a set of contracts rather than an individual contract. If X is a class, a contract such as in Figure 6 binds all instances of X. We will denote such a contract by $X.\mathsf{contract}$. If x is an object of class X, the specific instance of contract that binds x is denoted by $x.\mathsf{contract}$, which is obtained by replacing every variable v in $X.\mathsf{contract}$ with $x.v$ —assuming this does not cause any name collision.

6.1 Semantical Model

Formally, we will represent a contract c with a tuple of this type:

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

where $c.\mathsf{inv}$ 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's contractor. Properties specified in $c.\mathsf{progress}$ are intended to be robust: they prevail regardless the environment —we will return to this later. Furthermore we impose that $a.\mathsf{smodel}$ is a so-called *abstract object*, which is an object possessing no private variable[8] (so $a.\mathsf{smodel.pri} = \emptyset$).

[7]In a more general setup, a contract may only expose some abstractions of the operations. Just to keep the discussion simpler we will not do this here.

[8]This is not a principle restriction, but more a matter of choice in defining how expressive a contract should be.

We will also overload the selectors used on objects so that they also work on contracts, with their (the selectors') meaning unchanged. So for example, for an object x, $x.\mathsf{pub}$ denotes the the set of all x's public variables. With the same notation, if c is a contract, $c.\mathsf{pub}$ denotes the set of all c's public variables, which is just equal to $c.\mathsf{smodel.pub}$.

As said, $c.\mathsf{inv}$ specifies an invariant. More specifically, it is a strong invariant of $c.\mathsf{smodel}$, and of any proper environment of $c.\mathsf{smodel}$. Furthermore, $c.\mathsf{inv}$ has to be confined by $c.\mathsf{pub}$. Formally:

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

In the example in Figure 6, the `public` and `operation` sections of the corresponding object class, and the `safety model` of the contract form the smodel part; the `invariant` section forms the inv part; and the `progress` section forms the progress part.

We will now define the relation between an object and its contract:

Definition 6.1 : OBJECT-CONTRACT RELATION
Let $x :: X$ be an object and $c = x.\mathsf{contract}$. The relation between x and c is as follows:

1. x and $c.\mathsf{smodel}$ have the same interface. So $x.\mathsf{pub} = c.\mathsf{pub}$ and $x.\mathsf{ops} = c.\mathsf{ops}$.

2. There exists a predicate j such that:

   (a) j 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:

   $$j\ \vdash\ c.\mathsf{smodel} \sqsubseteq x$$

   (c) For every specification $p \mapsto q$ in $c.\mathsf{progress}$:

   $$x_{\lhd} \| x.\mathsf{env}, j\ \vdash\ p \mapsto q$$

□

The invariant j mentioned above is also called the *concrete invariant* of x, and will be denoted by $x.\mathsf{INV}$. Note an application developer cannot assume to know what this $x.\mathsf{INV}$ exactly is, though he knows that it has the properties as described in the object-contract relation above.

Note that since x and c have the same set of operations, then $x.\mathsf{env} = c.\mathsf{env}$. So, any proper environment of $c.\mathsf{smodel}$ is also a proper environment of x.

Note that although $c.\mathsf{inv}$ is a strong invariant of $c.\mathsf{smodel} \| c.\mathsf{env}$, it may not be a strong invariant of $x \| x.\mathsf{env}$. It is however an invariant of $x \| x.\mathsf{env}$, since the object-contract relation implies the existence of a strong invariant j of $x \| x.\mathsf{env}$ implying $c.\mathsf{inv}$.

6.2 Inferring Object's Properties

From the object-contract relation and from Theorem 4.9, it follow that the composition of x with any proper environment Q will maintain all progress properties specified in R:

Corollary 6.2 : Progress Commitment
Let $x :: X$ be an object and $c = x.\mathsf{contract}$. Let Q be a proper environment of x. Then, for any $p \mapsto q \in c.\mathsf{progress}$ we have:

$$x_{\triangleleft} [\![Q,\ x.\mathsf{INV} \ \vdash\ p \mapsto q$$

□

Any unless property proven with respect to the safety model in the contract is also, by Theorem 4.7, a property of the actual object. More precisely:

Theorem 6.3 : Safety Commitment
Let $x :: X$ be an object and $c = x.\mathsf{contract}$. Let Q be a proper environment of x. Then:

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

□

For example, (4) shows an example of an unless we can infer from the contract of `SimpleVotingSystem`. By the above theorem, it follows that the property holds for any `SimpleVotingSystem` object, under any proper environment.

As another example, the `progress` part of `SimpleVotingSystem`'s contract says that any valid vote in `votes` will eventually be copied to `validvotes`. Corollary 6.2 says now that this progress is guaranteed by any `SimpleVoting-System` object, under any proper environment.

Contract Refinement

One can also define a notion of contract refinement. Such a notion is useful when an object broker cannot find an object offering a contract c. In that case it may try to find another object whose contract refines c. We will not work out this idea further here.

7 Application

Figure 1 has shown an example of an application. An application consists of a set of objects and a set of programs, so-called *clients*, using the objects. An object can be a *private* object, it cannot be accessed by the application's environment, and otherwise it is a public object[9].

[9]In systems like CORBA a public object may actually be located outside the application itself. It may belongs to and controlled by another application, which may even be owned by a foreign organization. In such a setting a so-called *object broker* is used for searching the objects needed by an application and to facilitate the communication between the application and those foreign objects. This paper will however not concern itself with brokers.

7.1 Semantical Model

We will semantically model an application $\mathcal{A}$ with a tuple of this type:

$$\begin{array}{llllll} App & \overset{d}{=} & (& \mathsf{pubobjs} & :: & \{ObjDecl\} & , \\ & & & \mathsf{priobjs} & :: & \{ObjDecl\} & , \\ & & & \mathsf{clients} & :: & \{Prog_{\text{UNITY}}\} & \\ & & & \mathsf{clientsinv} & :: & Pred &) \end{array}$$

where *ObjDecl* represents object declarations. Its values are of form $x :: X$, where x is an object and X is the class to which x belongs. $\mathcal{A}.\mathsf{pubobjs}$ and $\mathcal{A}.\mathsf{privobjs}$ specify the set of $\mathcal{A}$'s, respectively, public and private objects. We will use the notation $\mathcal{A}.\mathsf{objs}$ to refer to the set of all objects of $\mathcal{A}$. The predicate $\mathcal{A}.\mathsf{clientsinv}$ is intended to augment the invariants of $\mathcal{A}$'s objects with information about the clients' (private) variables. In addition, there are several constraints, for example concerning the well-formedness of the model; we will show them later, because we need to introduce several concepts first.

The set of all public variables of $\mathcal{A}$, denoted by $\mathcal{A}.\mathsf{pub}$, consists of the public variables of its public objects. Its private variables, denoted by $\mathcal{A}.\mathsf{pri}$, consists of the private variables of its public objects, and all variables of its private objects and clients. The set of all $\mathcal{A}$'s variables is denoted by $\mathcal{A}.\mathsf{var}$.

The concrete program induced by an application $\mathcal{A}$ is the following:

$$\mathcal{A}.\mathsf{prg} \;\overset{d}{=}\; (\,[\!]x : x \in \mathcal{A}.\mathsf{objs} : x)\ [\!]\ (\,[\!]Q : Q \in \mathcal{A}.\mathsf{clients} : Q)$$

Of course, the application developer does not actually see the programs in $\mathcal{O}$ and $\mathcal{I}$. Instead, all he knows about those objects are their contracts. Every contract describes however a program abstracting the internal program of the contractor. We can construct the total abstract model of the objects in application by composing the safety models in their contracts. The total abstract model of the application, denoted by $\mathcal{A}.\mathsf{smodel}$, is the total abstract model of the objects, composed with the clients. Formally:

Definition 7.1 : Abstract Model of Application
Let $\mathcal{A}$ be an application. We define:

$$\begin{array}{lll} \mathcal{A}.\mathsf{model} & \overset{d}{=} & (\,[\!]x : x \in \mathcal{A}.\mathsf{objs} : x.\mathsf{contract.smodel}) \\ & & [\!] \\ & & (\,[\!]Q : Q \in \mathcal{A}.\mathsf{clients} : Q) \end{array}$$

□

The worst environment of an application can modelled by a program that tries all possible calls to the operations of the public objects:

Definition 7.2 : Application's Abstract Environment

$$\mathcal{A}.\mathsf{env} \;\overset{d}{=}\; (\,[\!]x, X : x :: X \in \mathcal{A}.\mathsf{pubobjs} : x.\mathsf{env})$$

□

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

Definition 7.3 : Application's Invariants
Let $\mathcal{A}$ be an application. We define:

$$\mathcal{A}.\mathsf{inv} \quad \overset{d}{=} \quad (\bigwedge x : x \in \mathcal{A}.\mathsf{objs} : x.\mathsf{contract.inv}) \ \wedge \ \mathcal{A}.\mathsf{clientsinv}$$

$$\mathcal{A}.\mathsf{INV} \quad \overset{d}{=} \quad (\bigwedge x : x \in \mathcal{A}.\mathsf{objs} : x.\mathsf{INV}) \ \wedge \ \mathcal{A}.\mathsf{clientsinv}$$

□

7.2 Constraints

As mentioned earlier there are a number of constraints we put on the semantical model of an application. Let $\mathcal{A}$ be an application. We have the following constraints:

1. [CA1] Each object in $\mathcal{A}$ has its own unique name space. So, for any two distinct objects x and y in $\mathcal{A}$, $x.\mathsf{var} \cap y.\mathsf{var} = \emptyset$.

   Note this implies that objects inside an application cannot call each other operations. In other words, we require each object to be an independent component. This simplifies the formal model[10].

2. [CA2] A client can only interact with an object through its operations. Furthermore, a client can only do a single operation in one atomic step. In other words, with respect to every object x in $\mathcal{A}$, each client should be a proper environment of x.

3. [CA3] The only public information a client has access to is the set of all public variables of the objects in $\mathcal{A}$. So, for every $Q \in \mathcal{A}.\mathsf{clients}$:

   $$Q.\mathsf{pub} \ \subseteq \ (\bigcup x : x \in \mathcal{A}.\mathsf{objs} : x.\mathsf{pub})$$

4. [CA4] $\mathcal{A}.\mathsf{clientsinv}$ can only specify the variables known to the clients:

   $$\mathcal{A}.\mathsf{clientsinv} \quad \textbf{conf} \quad \mathcal{A}.\mathsf{model.var}$$

   Note that this constraint is actually implied by the following one.

5. [CA5] $\mathcal{A}.\mathsf{clientsinv}$ can be maintained by $\mathcal{A}$'s abstract model:

   $$\mathcal{A}.\mathsf{model} [] \mathcal{A}.\mathsf{env}, \ i \ \vdash \ \mathcal{A}.\mathsf{clientsinv} \ \mathsf{unless\ false}$$

   where $i = (\bigwedge x : x \in \mathcal{A}.\mathsf{objs} : x.\mathsf{contract.inv})$.

[10]There may however be a situation where one wants to build an object from smaller objects. This is not directly allowed in our framework, because the top level object will then need to access the name space of its sub-objects, which is forbidden by CA1. Instead, one can first build an application, which does have access to its objects' name space, and then wrap the application as an object —see Subsection 7.7.

7.3 Some Facts

We list here a number of facts about applications. The first one below states that we can indeed use $\mathcal{A}.\mathsf{model}$ as $\mathcal{A}$'s abstraction:

Theorem 7.4 : $\mathcal{A}.\mathsf{INV} \;\vdash\; \mathcal{A}.\mathsf{model} \sqsubseteq \mathcal{A}.\mathsf{prg}$

> **proof:** Since $\mathcal{A}.\mathsf{model}$ consists of $\mathcal{A}$'s clients and the safety models of $\mathcal{A}$'s objects, it suffices to show that for every object x in $\mathcal{A}$, its safety model ($x.\mathsf{contract.smodel}$) is an abstraction of $x.\mathsf{prg}$. But this already follows from the object-contract relation.
> **end**

Theorem 7.5 :
Let x be an object in $\mathcal{A}$. Let $(\mathcal{A} - x).\mathsf{prg}$ denote the program obtained by removing the actions of $x.\mathsf{prg}$ from $\mathcal{A}.\mathsf{prg}$[11]. We have:

$$i \vdash x.\mathsf{env} \;\sqsubseteq\; ((\mathcal{A} - x).\mathsf{prg} \| \mathcal{A}.\mathsf{env})$$

for any i. $\Box$

> **proof:** (1) Each action a in $(\mathcal{A} - x).\mathsf{prg}$ is either an action from another object y or an action of a client program Q. If it is an action from another object, then it behaves as skip with respect to $x.\mathsf{var}$, because each object has its own unique name space ($\mathsf{CA1}$), and hence it automatically (weakly) refines $x.\mathsf{env}$. If a is an action of Q, it too refines $x.\mathsf{env}$, because each client is proper environment with respect to each object ($\mathsf{CA2}$).
>
> (2) Each action in $\mathcal{A}.\mathsf{env}$ is either a call to an operation of another object y, or a call to an operation of x. By similar argument as above, in both cases it too will refine $x.\mathsf{env}$.
> **end**

Theorem 7.6 : $\mathcal{A}.\mathsf{INV} \;\Rightarrow\; \mathcal{A}.\mathsf{inv}$

> **proof:** It follows from their definition and the fact that for any object x, $x.\mathsf{INV}$ implies $x.\mathsf{contract.inv}$.
> **end**

The theorem below says that $\mathcal{A}.\mathsf{inv}$ is a strong invariant of both $\mathcal{A}$'s abstract model and its environment.

Theorem 7.7 : $\mathcal{A}.\mathsf{model} \| \mathcal{A}.\mathsf{env} \;\vdash\; \mathsf{sinv}\; \mathcal{A}.\mathsf{inv}$

> **proof:** Let $i = (\bigwedge x : x \in \mathcal{A}.\mathsf{objs} : x.\mathsf{contract.inv})$. Note that:
>
> (a) $\quad \mathcal{A}.\mathsf{inv} \;= i \;\wedge \mathcal{A}.\mathsf{clientsinv}$

[11] The set of variables and the initial condition are left unchanged.

Each x.contract.inv part of i is a strong invariant of x.smodel$\|x$.env. It is also maintained by y.smodel$\|y$.env, for any other object y, because objects do not share name space (CA1). The only way the clients interact with x is through its operations (CA2), and thus they refine x.env; hence they too maintain x.contract.inv. Hence, x.contract.inv is a strong invariant in $\mathcal{A}$.model$\|\mathcal{A}$.env. Because this holds for all $x \in \mathcal{A}$.objs, it follows, by the definition of i:

(b) $\mathcal{A}$.model$\|\mathcal{A}$.env $\vdash$ sinv i

The invariance of $\mathcal{A}$.inv follows from (a), (b), and CA5.
end

The theorem below is the analogous of the one above for the application's concrete invariant:

Theorem 7.8 : $\mathcal{A}$.prg$\|\mathcal{A}$.env $\vdash$ sinv $\mathcal{A}$.INV

proof: The proof is quite similar to that of Theorem 7.7. Let $j = (\bigwedge x : x \in \mathcal{A}$.objs : x.INV). Note that:

(a) $\mathcal{A}$.INV $= j \wedge \mathcal{A}$.clientsinv

Each x.INV part of i is by definition a strong invariant of $x\|x$.env. It is also maintained by $y\|y$.env, for any other object y, because objects do not share name space (CA1). The only way the clients interact with x is through its operations (CA2), and thus they refine x.env; hence they too maintain x.INV. Hence, x.INV is a strong invariant in $\mathcal{A}$.prg$\|\mathcal{A}$.env. Because this holds for all $x \in \mathcal{A}$.objs, it follows, by the definition of j:

(b) $\mathcal{A}$.prg$\|\mathcal{A}$.env $\vdash$ sinv j

Let i be defined as in CA5. Note that $j \Rightarrow i$. Next, by CA5, Theorem 7.4, $j \Rightarrow i$, and Theorem 4.7 we have:

(c) $\mathcal{A}$.prg$\|\mathcal{A}$.env, j $\vdash$ $\mathcal{A}$.clientsinv unless false

The invariance of $\mathcal{A}$.INV follows now from (a), (b), and (c).
end

7.4 Inferring an Application's Properties

The following theorem says that a safety property proven with respect to the abstract model of an application will extend to the application itself, under any proper environment.

Theorem 7.9 : Safety by Abstract Model

$$\frac{\mathcal{A}.\mathsf{model} \| \mathcal{A}.\mathsf{env},\ \mathcal{A}.\mathsf{inv}\ \vdash\ p\ \mathsf{unless}\ q}{\mathcal{A}.\mathsf{prg} \| \mathcal{A}.\mathsf{env},\ \mathcal{A}.\mathsf{INV}\ \vdash\ p\ \mathsf{unless}\ q}$$

□

proof: First:

1. From Theorem 7.8 we have that $\mathcal{A}.\mathsf{INV}$ is a strong invariant of $\mathcal{A}.\mathsf{prg} \| \mathcal{A}.\mathsf{env}$
2. By Theorem 7.6: $\mathcal{A}.\mathsf{INV} \ \Rightarrow\ \mathcal{A}.\mathsf{inv}$.
3. By Theorem 7.4 we have that $\mathcal{A}.\mathsf{prg}$ refines $\mathcal{A}.\mathsf{model}$, with respect to $\mathcal{A}.\mathsf{INV}$.

From the above, and Theorem 4.7 the premise of the above theorem now follows from its assumption.
end

The next theorem says that any progress property committed in the contract of any object in an application, will be preserved by the application, and by its environment.

Theorem 7.10 : Progress by Contract
Let $x :: X$ be an object in $\mathcal{A}$ and $c = x.\mathsf{contract}$.

$$\frac{p \mapsto q\ \in\ c.\mathsf{progress}}{\mathcal{A}.\mathsf{prg}_{\lhd} \| \mathcal{A}.\mathsf{env},\ \mathcal{A}.\mathsf{INV}\ \vdash\ p \mapsto q}$$

□

proof: Let $(\mathcal{A} - x).\mathsf{prg}$ denote the program obtained by removing the actions of $x.\mathsf{prg}$ from $\mathcal{A}.\mathsf{prg}$ (as in Theorem 7.5). Let $rest = (\mathcal{A} - x).\mathsf{prg} \| \mathcal{A}.\mathsf{env}$. Note that $x.\mathsf{prg} \| rest = mathcalA.\mathsf{prg} \| \mathcal{A}.\mathsf{env}$. We derive now:

$p \mapsto q\ \in\ c.\mathsf{progress}$

$\Rightarrow$ { object-contract relation (Definition 6.1) }

$x.\mathsf{prg}_{\lhd} \| x.\mathsf{env},\ x.\mathsf{INV}\ \vdash\ p \mapsto q$

$\Rightarrow$ { (1) by Theorems 7.5 we have: $\mathcal{A}.\mathsf{INV} \vdash x.\mathsf{env} \sqsubseteq rest$; (2) by Theorem 7.8: $\mathcal{A}.\mathsf{INV}$ is a strong invariant of $x.\mathsf{prg} \| rest$; (3) by definition, $\mathcal{A}.\mathsf{INV}$ implies $x.\mathsf{INV}$; (4) then using Theorem 4.9 }

$x.\mathsf{prg}_{\triangleleft} [\![rest, \ \mathcal{A}.\mathsf{INV} \ \vdash \ p \mapsto q$

$\Rightarrow$ { def. of *rest* and Theorem A.10 }

$(x.\mathsf{prg} [\![(\mathcal{A} - x).\mathsf{prg})_{\triangleleft} [\![\mathcal{A}.\mathsf{env}, \ \mathcal{A}.\mathsf{INV} \ \vdash \ p \mapsto q$

$=$ { by definition of $A - x$ }

$\mathcal{A}.\mathsf{prg}_{\triangleleft} [\![\mathcal{A}.\mathsf{env}, \ \mathcal{A}.\mathsf{INV} \ \vdash \ p \mapsto q$

end

The theorem below states the kind of progress made by the clients that can be be preserved by the application and its environment.

Theorem 7.11 : Client Progress
Let:

$$\begin{array}{lcl} clients & = & ([\![Q : Q \in \mathcal{A}.\mathsf{clients} : Q) \\ smodel & = & ([\![x : x \in \mathcal{A}.\mathsf{objs} : x.\mathsf{contract.smodel}) \end{array}$$

We have:

$$\frac{clients_{\triangleleft} [\![(smodel [\![\mathcal{A}.\mathsf{env}), \ \mathcal{A}.\mathsf{inv} \ \vdash \ p \mapsto q}{\mathcal{A}.\mathsf{prg}_{\triangleleft} [\![\mathcal{A}.\mathsf{env}, \mathcal{A}.\mathsf{INV} \ \vdash \ p \mapsto q}$$

□

proof:

$clients_{\triangleleft} [\![(smodel [\![\mathcal{A}.\mathsf{env}), \ \mathcal{A}.\mathsf{inv} \ \vdash \ p \mapsto q$

$\Rightarrow$ { Theorem A.10, $clients [\![smodel = \mathcal{A}.\mathsf{model}$ }

$\mathcal{A}.\mathsf{model}_{\triangleleft} [\![\mathcal{A}.\mathsf{env}, \ \mathcal{A}.\mathsf{inv} \ \vdash \ p \mapsto q$

$\Rightarrow$ { (1) by Theorems 7.4: $\mathcal{A}.\mathsf{model} \ \sqsubseteq \ \mathcal{A}.\mathsf{prg}$; (2) by Theorem 7.8: $\mathcal{A}.\mathsf{INV}$ is invariant; (3) by Theorem 7.6: $\mathcal{A}.\mathsf{INV}$ implies $\mathcal{A}.\mathsf{inv}$; (4) then using Theorem 4.9 }

$\mathcal{A}.\mathsf{prg}_{\triangleleft} [\![\mathcal{A}.\mathsf{env}, \ \mathcal{A}.\mathsf{INV} \ \vdash \ p \mapsto q$

end

7.5 Example

Consider again the `VotingService` application in Figure 1. The application uses an object of class `SimpleVotingSystem`. Recall the property (1), saying once the voting process is closed, no new vote will be accepted. This is however not a property of an application, but merely a property of any object `x` of the class `SimpleVotingSystem`. We want now to infer this property for our application. Formally, this is the property we want to show:

$$\mathsf{app.prg} [\![\mathsf{app.env}, \ j \ \vdash \ \neg \mathtt{v.isOpen} \wedge v \not\in \mathtt{v.votes} \ \ \mathsf{unless} \ \ \mathsf{false} \tag{5}$$

where $\mathsf{app} = \mathtt{VotingService}$ and $j = \mathsf{app.INV}$. By Theorem 7.9 it is sufficient to show the following:

$$\mathsf{app.model} \| \mathsf{app.env},\ i\ \vdash\ \neg\mathtt{v.isOpen} \wedge v \notin \mathtt{v.votes}\ \ \mathsf{unless}\ \ \mathsf{false} \tag{6}$$

where $i = \mathsf{app.inv}$ The only action in $\mathsf{app.model} \| \mathsf{app.env}$ that can violate this property is the operation `vote`, because it can insert v to `votes`. However, it can only do so if `isOpen` is true, which is not the case in the unless property above. Hence, it too respects the above property. Hence, the above property holds in $\mathsf{app.model} \| \mathsf{app.env}$.

In the next example let us consider this progress property: if there are already more than 100 votes in the list `validvotes`, then eventually the value of `accept` will be set to true[12]:

$$\mathsf{app.prg}_{\triangleleft} \| \mathsf{app.env},\ j\ \vdash\ \mathtt{length}\ \mathtt{v.validvotes} > 100 \mapsto \mathtt{v.accept} \tag{7}$$

We will show the property with the help of several smaller properties of $\mathsf{app.prg}_{\triangleleft} \| \mathsf{app.env}$, with respect to the invariant j:

$$\mathsf{true}\ \ \mapsto\ \ \mathtt{d.current} \geq \mathtt{closingdate} \tag{8}$$

$$\mathtt{d.current} \geq \mathtt{closingdate}\ \ \mapsto\ \ afterClosingDate \tag{9}$$

where $afterClosingDate = \mathtt{d.current} \geq \mathtt{closingdate} \wedge \mathtt{today} \geq \mathtt{closingdate}$

$$\mathtt{length}\ \mathtt{v.validvotes} > 100\ \mathsf{unless}\ \mathsf{false} \tag{10}$$

$$\mathtt{length}\ \mathtt{v.validvotes} > 100\ \wedge\ afterClosingDate\ \ \mapsto\ \ \mathtt{v.accept} \tag{11}$$

The combination of (8), (9), and (10) implies that from $\mathtt{length}\ \mathtt{v.validvotes} > 100$ the application can progress to:

$$\mathtt{length}\ \mathtt{v.validvotes} > 100\ \wedge\ afterClosingDate$$

Subsequently, by (11) it will further progress to `v.accept`, and thus (7) is proven.

We are not going to prove all properties in (8) ... (11). The approach to prove (10) is similar to that of (5), and that of (11) similar to that of (9). So, we are going to show only the proofs of (8) and (9).

Consider now (8). Anticipating that this progress is realized by the `SimpleCalendar` object of app, we use Theorem 7.10 to reduce the application level property to an object level one. So, by the Theorem it suffices to show that the same property is specified in the `progress`-part of the object `d`, which is indeed the case (see the contract in Figure 1.

Consider now (9). We expect this progress to be realized by the client. So, we use Theorem 7.11. By the Theorem, it suffices if we can show the

[12]We can also show a stronger property, namely that if there are already more than 100 incoming valid votes (either still in `votes` and already in `validvotes`), then eventually `accept` will be `true`. The proof of this is however more involved, so for the example we choose a simpler property.

same progress, but with respect to $\mathsf{app.model}_{\triangleleft}[\!]\mathsf{app.env}$ and invariant i. The specified progress is a simple one, realized by the act of a single action, namely the action `d.getDate(today)` of the client. Therefore, it suffices to show the following $\mathsf{ensures}$ property, which implies $\mapsto$:

$$\begin{array}{l}\mathsf{app.model}_{\triangleleft}[\!]\mathsf{app.env},\ \mathtt{i} \\ \vdash \\ \mathtt{d.current} \geq \mathtt{closingdate} \quad \mathsf{ensures} \quad \mathit{afterClosingDate}\end{array} \tag{12}$$

This property can be quite easily proven.

7.6 Open and Closed System

An application with non-empty set of public objects represents an open system with which any other system can interact through operations of the public objects. A closed system can be represented by an application that exposes no public object. For both kinds of systems, we can use the same theory for applications in Section 7.

7.7 Deploying an Application as an Object

Since an application is essentially just a program, it can be deployed as an object. First we have to define an interface, because an object has public and private variables, initial condition, and operations. An obvious choice is to expose all public variables and operations of the public objects of an application as the public variables and operations of the resulting object. However, we may also want to expose less. For example, consider again the `votingService` application from Figure 1. This application still has a problem, namely that an arbitrary user can call the `count` method of `v`, forcing the votes to be counted before the closing date. A way to get around this is not to give the users access to the operation `count`. This is achieved by wrapping the application `votingService`, for example by writing:

```
wrap votingService
   expose d
   expose v hiding count
```

The code above wraps the application `votingService` to an object. It exposes all public variables and operations of `d`. It also exposes `v`, but it hides the operation `vote` of `v`.

A wrapper can only expose public objects of an application. Since wrapping may hide some part of a public object, this can only be done safely if the object is fully controlled by the application itself (thus not an external object used by the application).

If we require that wrapping can only hide variables and operations, and not to introduce new ones, it follows that the actual environment of a wrapped application $\mathcal{A}$ will not behave worse that $\mathcal{A}.\mathsf{env}$. Consequently, properties we can infer from $\mathcal{A}$ using the theorems from Subsection 7.4, which assume $\mathcal{A}.\mathsf{env}$, will be preserved by the wrapping.

8 Related Works

The area of component based systems is currently very actively researched. We will only mention several recent related works. Some examples of other formal frameworks for component based systems are: Kim and Carrington's [11], Jifeng, Liu, and Xiaoshan's [10], and Broy's [4]. The first one, based on Z, focuses on formalizing static relations between components, specified in terms of UML class diagrams. The other two, like ours, focus on the behavioural aspect of systems and their components. Jifeng, Liu, and and Xiaoshan's framework relies on Hoare triple specifications to infer behavioural properties. It is suitable to deal with components that have no internal programs of their owns, and hence cannot, on their own, enforce temporal properties. If internal programs are added, specifying temporal properties would require the use of auxiliary variables to record the objects' history. Broy's framework has a built in history variables, which are already part of its logic. The framework is especially tailored for dealing with components that synchronize with channels. It may however be too detailed if the components exchange information through operation calls instead. In contrast, our framework is more attuned to the latter scenario.

An important part in our framework is the use of an abstract program to specify (a certain aspect of) the behavior of a concrete program or environment. It turns out to be quite convenient. The same idea has been voiced by many other researches, for example by Collete and Knapp [8, 9], Udink [19], Shankar [17], and by Jifeng, Lui, and Xiaoshan [10].

We favour the use of UNITY as the underlying theory, because of its simplicity and its axiomatic style. It yields a semantical model which is simple and intuitive. The use of UNITY as the underlying theory to support component based design has also been proposed by other researchers, for example by Collete and Knapp [8, 9] and Udink [19]. As far as we know, our work is the first UNITY framework offering formal notions of objects, contracts, and application.

A key part in any formal system about component based systems is the ability to compositionally infer a property of a system from the properties of its components. The original UNITY [5] does not support compositional inference of progress properties. People use extensions to get around this. Examples of such compositional extensions can be found in Collete and Knapp's and Udink's works mentioned above. The extension we use originates from our previous work in [15]. It is more general than [15]. It conveys the same concept as Collete and Knapp's extension [9], though the semantical model is quite different. Fundamental theories about program composition, in particular in distributed setting, can be found in Abadi and Lamport's work [1, 2] and Chandy, Sanders, and Charpentier's [6, 7].

9 Conclusion

We have offered a formal framework to support a component based approach to build distributed applications. The framework offers formal notions of objects, contracts, and applications, and a set of laws to compositionally infer temporal properties of an application.

The underlying theory is in UNITY style, which, for example, compared to LTL is indeed less expressive. However, because of UNITY's axiomatic style, it yields a semantical model which is simple and intuitive.

The small experiment with the simple voting system example shows that contracts in our framework can adequately, abstractly, and conveniently capture the system's components' temporal properties. Using the laws provided by the framework we are able to compositionally infer interesting properties of the system, using basically only the information provided by the components' contracts.

We have chosen for a lenient notion of refinement, which gives more freedom to developers in hiding their components' details in the contracts. It is possible to take a stronger notion of refinement, for example as in [20, 21], but in exchange checking if a component satisfies its contract will be more expensive. Our framework is suitable for dealing with systems in which components synchronize by operations calls. It is less suitable to deal with message passing systems, or with systems where components require tight synchronization as in protocols.

Overall, we believe that the framework is at least worth further investigation. In the future we would like to experiment with larger examples, to see its scalability. On the positive result we would then like to develop an actual user-level programming and specification language (a real Dalang language, not a hypothetical one, as used here) and its tool support (like a compiler, verification condition generator, etc), and thus bringing the theory into practice.

A Additional Laws

Theorem A.1 : Privacy
Let W be a set of variables. Let $\overline{w}$ denote the variables in W listed as a vector. Let $e[\overline{val}/\overline{w}]$ denote the result of replacing every free occurence of w_i in e with val_i.

If b is an action that does not write to W (but may read from it), we have:

$$\{i \wedge p\}\ b\ \{q\} \quad = \quad (\forall \overline{val} :: \{i \wedge p[\overline{val}/\overline{w}] \wedge (\overline{w} = \overline{val})\}\ b\ \{q[\overline{val}/\overline{w}] \wedge (\overline{w} = \overline{val})\})$$

If b is an action that does not read from nor write to any variable in W, we have:

$$\{i \wedge p\}\ b\ \{q\} \quad = \quad (\forall \overline{val} :: \{i \wedge p[\overline{val}/\overline{w}]\}\ b\ \{q[\overline{val}/\overline{w}]\})$$

□

proof: We will prove the case when b does not write to W first:

$\{i \wedge p\}\ b\ \{q\}$

$=$ { b does not write to W;disjunction of Hoare triple }

$(\forall \overline{val} ::\ \{i \wedge p \wedge (\overline{w} = \overline{val})\}\ b\ \{q \wedge (\overline{w} = \overline{val})\})$

$=$ { trivial }

$(\forall \overline{val} ::\ \{i \wedge p[\overline{val}/\overline{w}] \wedge (\overline{w} = \overline{val})\}\ b\ \{q[\overline{val}/\overline{w}] \wedge (\overline{w} = \overline{val})\})$

For the second case, where b does not read from W either, notice that the last formula in the derivation above, because b does not read from W, is equivalent to:

$(\forall \overline{val} ::\ \{i \wedge p[\overline{val}/\overline{w}]\}\ b\ \{q[\overline{val}/\overline{w}]\})$

end

Theorem A.2 :
If a and b do not read from nor write to any variables in W, then:

$$\frac{V, i \vdash a \leq b}{V \cup W, i \vdash a \leq b}$$

proof: It suffices to show the following for all p, q **conf** $V \cup W$:

$\{i \wedge p\}\ b\ \{q\}$

$=$ { Theorem A.1 }

$(\forall \overline{val} ::\ \{i \wedge p[\overline{val}/\overline{w}]\}\ b\ \{q[\overline{val}/\overline{w}]\})$

$\Leftarrow$ { p, q **conf** $V \cup W$, hence $p[\overline{val}/\overline{w}], q[\overline{val}/\overline{w}]$ **conf** V; the refinement relation in the premise }

$(\forall \overline{val} ::\ \{i \wedge p[\overline{val}/\overline{w}]\}\ a\ \{q[\overline{val}/\overline{w}]\})$

$=$ { Theorem A.1 }

$\{i \wedge p\}\ a\ \{q\}$

end

Theorem A.3 : skip Non-observability
For any V, and W, the action $\mathsf{skip}\ V$ does not read from nor write to the variables in W.
□

Theorem A.4 :

$$\frac{V, i \vdash a \sqcup \mathsf{skip}\ V \leq b}{V, i \vdash a \sqcup \mathsf{skip}\ (V \cup W) \leq b}$$

□

proof: It suffices to prove the following for all p, q **conf** V:

$\{i \wedge p\}\ b\ \{q\}$

$\Leftarrow$ { the refinement in the premise }

$\{i \wedge p\}\ a \sqcup \mathsf{skip}\ V\ \{q\}$

$\Leftarrow$ { p, q **conf** V }

$\{i \wedge p\}\ a \sqcup \mathsf{skip}\ (V \cup W)\ \{q\}$

end

Theorem A.5 :
If a and b do not read from nor write to any variables in W, then:

$$\frac{V, i \vdash a \sqsubseteq b}{V \cup W, i \vdash a \sqsubseteq b}$$

proof:

$V \cup W, i \vdash a \sqsubseteq b$

$=$ { Definition of weak refinement }

$V \cup W, i \vdash a \sqcup \mathsf{skip}\ (V \cup W) \leq b$

$\Leftarrow$ { a does not read from and write to W; by Theorem A.3 neither does $\mathsf{skip}\ V$; then apply Theorem A.2 }

$V, i \vdash a \sqcup \mathsf{skip}\ (V \cup W) \leq b$

$\Leftarrow$ { Theorems A.4 }

$V, i \vdash a \sqcup \mathsf{skip}\ V \leq b$

$=$ { Definition of weak refinement }

$V, i \vdash a \sqsubseteq b$

end

Theorem A.6 :

$$\frac{V, i \vdash a \leq c}{V, i \vdash a \sqcup b \leq c}$$

$$\frac{V, i \vdash a \sqsubseteq c}{V, i \vdash a \sqcup b \sqsubseteq c}$$

$\square$

proof: For the first law:

$\{i \wedge p\}\ c\ \{q\}$

$\Leftarrow$ { refinement in the premise }

$\{i \wedge p\}\ a\ \{q\}$

$\Leftarrow$ { definition of $\sqcup$ }

$\{i \wedge p\}\ a \sqcup b\ \{q\}$

For the second one:

$V, i \vdash a \sqcup b \sqsubseteq c$

$=$ { definition of $\sqsubseteq$ }

$V, i \vdash a \sqcup b \sqcup \mathsf{skip}\ V \leq c$

$\Leftarrow$ { the first law above }

$V, i \vdash a \sqcup \mathsf{skip}\ V \leq c$

$=$ { definition of $\sqsubseteq$ }

$V, i \vdash a \sqsubseteq c$

end

Theorem A.7 : $V, i \vdash b \sqsubseteq b$

Theorem A.8 : Extending V in Refinement

$$\frac{V, i \mid\!\!-\!\!- A \sqsubseteq P \quad , \quad W \cap A.\mathsf{var} = W \cap P.\mathsf{var} = \emptyset}{V \cup W, i \mid\!\!-\!\!- A \sqsubseteq P}$$

$\square$

proof: Note first that the second premise implies that neither A nor P reads from nor writes to W. We derive now:

$V \cup W, i \mid\!\!-\!\!- A \sqsubseteq P$

$=$ { definition of program refinement }

$(\forall b : b \in P.\mathsf{acts} : V \cup W, i \vdash \sqcup A.\mathsf{acts} \sqsubseteq b)$

$\Leftarrow$ { $\sqcup A.\mathsf{acts}$ nor b reads from and writes to W; Theorem A.5 }

$(\forall b : b \in P.\mathsf{acts} : V, i \vdash \sqcup A.\mathsf{acts} \sqsubseteq b)$

$=$ { definition of program refinement }

$V, i \mid\!\!-\!\!- A \sqsubseteq P$

end

Theorem A.9 :

$$\frac{V, i \mid\!\!-\!\!- A \sqsubseteq P}{V, i \mid\!\!-\!\!- A \| Q \sqsubseteq P \| Q}$$

□

proof: We have to show this:

$V, i \vdash \sqcup((A\|Q).\mathsf{acts}) \sqsubseteq b$

for all actions $b \in (P\|Q).\mathsf{acts}$. There are two cases: b is an action of Q or b is an action of Q.

Case 1: Let b be an action of Q. We derive now:

$V, i \vdash \sqcup((A\|Q).\mathsf{acts}) \sqsubseteq b$

$=$ { definition $\|$ and $\sqcup$ }

$V, i \vdash ((\sqcup A.\mathsf{acts}) \sqcup (\sqcup Q.\mathsf{acts})) \sqsubseteq b$

$\Leftarrow$ { Theorem A.6 }

$V, i \vdash \sqcup Q.\mathsf{acts} \sqsubseteq b$

$\Leftarrow$ { $b \in Q.\mathsf{acts}$, Theorem A.6 }

$V, i \vdash b \sqsubseteq b$

$\Leftarrow$ { Theorem A.7 }

true

Case 2: Let b be an action of P. We derive now:

$V, i \vdash \sqcup((A \| Q).\mathsf{acts}) \sqsubseteq b$

$=$ { definition $\|$ and $\sqcup$ }

$V, i \vdash ((\sqcup A.\mathsf{acts}) \sqcup (\sqcup Q.\mathsf{acts})) \sqsubseteq b$

$\Leftarrow$ { Theorem A.6 }

$V, i \vdash \sqcup A.\mathsf{acts} \sqsubseteq b$

$\Leftarrow$ { $b \in P.\mathsf{acts}$ }

$V, i \mid\!\!-\!\!- A \sqsubseteq P$

end

Theorem A.10 : Left Shifting of $_{\triangleleft}\|$

$$\frac{P_{\triangleleft}\|(Q\|R), i \ \vdash \ p \mapsto q}{(P\|Q)_{\triangleleft}\|R, i \ \vdash \ p \mapsto q}$$

□

proof: This is proven using $\mapsto$ induction. The transitivity and disjunctivity cases are trivial. So, it suffices to show the ensures case:

$P_{\triangleleft}\|(Q\|R), i \vdash p$ ensures q

$=$ { Definition 4.8, $\|$ is associative }

$(P\|Q)\|R, \ i \ \vdash \ p$ unless $q \wedge (\exists a : a \in P.\mathsf{acts} : \{i \wedge p \wedge \neg q\} \ a \ \{q\})$

$\Rightarrow$ { trivial }

$(P\|Q)\|R, \ i \ \vdash \ p$ unless $q \ \wedge \ (\exists a \ : \ a \in P\|Q.\mathsf{acts} \ : \ \{i \wedge p \wedge \neg q\} \ a \ \{q\})$

$=$ { Definition 4.8 }

$(P\|Q)_{\triangleleft}\|R, j \vdash p \mapsto q$

end

Theorem A.11 : Enlarging Interference Scope

$$\frac{P, i \ \vdash \ p \text{ unless } q \text{ using } V \quad , \quad V \subseteq W}{P, i \ \vdash \ p \text{ unless } q \text{ using } W}$$

□

proof: By the definition of unless it suffices to show p, q **conf** W. The theorem's premise implies that p and q are confined by V. Since W is a superset of V, it too confines p and q.
end

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[3] R.J.R. Back and J. Von Wright. Refinement calculus, part I: Sequential non-deterministic programs. 430:42–66, 1989.

[4] M. Broy. Multi-view modelling of software sytems. In Hung Dang Van and Zhiming Liu, editors, *Proceedings of the Workshop on Formal Aspects of Component Software (FACS)*, 2003. Also as UNU/IIST Report no. 284, available on-line at `www.iist.unu.edu/newrh/III/1/page.html`.

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

[6] K.M. Chandy and B.A. Sanders. Reasoning about program composition. Draft. Presently available via: `www.cise.ufl.edu/~sanders/pubs`, 2000.

[7] M. Charpentier and K. Chandy. Theorems about composition. 1837:167–186, 2000.

[8] P. Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, December 1994.

[9] P. Collette and E. Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. 936:353 – 367, 1995.

[10] He Jifeng, Lui Zhiming, and Li Xiaoshan. A contract-oriented approach to CBP. In Hung Dang Van and Zhiming Liu, editors, *Proceedings of the Workshop on Formal Aspects of Component Software (FACS)*, 2003. Also as UNU/IIST Report no. 284, available on-line at `www.iist.unu.edu/newrh/III/1/page.html`.

[11] Soon-Kyeong Kim and David Carrington. A formal mapping between UML models and Object-Z specifications. *Lecture Notes in Computer Science*, 1878:2–??, 2000.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Specification.* Springer-Verlag, 1992.

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

[14] I.S.W.B. Prasetya. Error in the UNITY substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.

[15] I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra, and B. Widjaja. A theory for composing distributed components based on temporary interface. In Hung Dang Van and Zhiming Liu, editors, *Proceedings of the Workshop on Formal Aspects of Component Software (FACS)*, 2003. Also as UNU/IIST Report no. 284. Available on-line at `www.iist.unu.edu/newrh/III/1/page.html`.

[16] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[17] N. Shankar. Lazy compositional verification. 1536:541–564, 1999.

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

[19] R.T. Udink. *Program Refinement in UNITY-like Environments.* PhD thesis, 1995. Downloadable from `www.cs.uu.nl`.

[20] T.E.J. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms.* PhD thesis, 2000. Download: `www.cs.uu.nl`.

[21] 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.