∀UNITY: A Theory of General UNITY

I.S.W.B. Prasetya, T.E.J. Vos, A. Azurat, S.D. Swierstra
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands
Instituto Tecnológico de Informática, Universidad Politécnica de Valencia
Camino de Vera s/n, 46071 Valencia, Spain
{*wishnu,tanja,doaitse,ade*}@*cs.uu.nl*

Abstract

UNITY is a simple programming logic to reason about distributed systems. It is especially attractive because of its elegant axiomatical style. Since its power is limited, people introduce variants to extends it with various new functionalities. However, in the axiomatical style it is easy to make a mistake: a seemingly very logical new inference rule may turn out to be unsound. Formal verification is often necessary, but it is a time consuming task. ∀UNITYis a generalization of UNITY. It provides the same set of inference rules, but they are now derived from much more primitive (weaker) rules. ∀UNITYis provided as a HOL (a theorem prover) library, with all its derived rules mechanically verified. Using ∀UNITYa sound and complete UNITY variant (instance) can be quickly created by showing that the instance upholds ∀UNITYprimitive rules. Moreover, all theories one subsequently derives from ∀UNITYwill be valid for all ∀UNITYinstances.

1 Introduction

UNITY is a programming language and an associated logic introduced by Chandy and Misra in 1988 to reason about distributed systems. The logic consists of three operators, unless, ensures, and $\mapsto$ (leads-to), with which we can specify temporal properties of a distributed system. Moreover, there are many inference rules that can be used to prove these properties. It is a simple logic, because it restricts itself to first order temporal properties, which for many applications are sufficient.

UNITY is excellent for the formal treatment of distributed algorithms, e.g. as in [10, 17, 21, 22]. Distributed algorithms algorithms tend to be abstract, subtle, and parameterized with higher order information –a combination that makes them quite beyond the reach of automated deduction and model checking.

UNITY is less suitable for automated program verification, since $\mapsto$ (progress) properties may be undecidable[1], though it is possible to map UNITY progress properties to similar properties in other formalisms where automated deduction or model checking is possible.

The simplicity of UNITY is often an advantage since less creativity is needed when constructing a proof. Moreover, it is easier to implement (mechanize) and provide a computer assisted proof environment. An example of a UNITY implementation can be found in a tool called xMECH [2] where UNITY is used as the base logic for a more abstract programming language. Another way of implementing UNITY is by representing it in a theorem prover, e.g. as done in [1, 16, 8], this gives an extra benefit since the implementation is guaranteed to be sound.

[1]That is, a proof that exhaustively applying UNITY inference rules may be non-terminating. Human insight is often needed to select which rules to apply at certain steps.

Although simplicity is an advantage, quite often we are confronted with proofs that require inference rules which are just beyond the power of UNITY. For example, classical UNITY misses a very useful inference rule called the *Substitution Law* [19] and rules to preserve $\mapsto$ properties in parallel composition [20, 18]. As a consequence, varios people have introduced variants [19, 14, 6, 5, 12, 18] to add new functionality to UNITY.

In the past, introducing a new logic was done with a lot of precaution. In truth, however, writing a new logic is no more magical than writing a program. When confronted with new kinds of problems, it makes sense to devise a new logic to more efficiently solve them. For example, one might want to have a variant of UNITY which is extended with probabilistic reasoning or with cryptographic reasoning. However, we have to be careful when writing a new logic or extending an existing one, especially when adapting an axiomatical approach. The axiomatical approach encourages us to capture the way we reason about things abstractly using inference rules. As opposed to the semantical (operational) style, it tends to produce a cleaner and more abstract logic. It is however quite easy to make mistakes: a seemingly very logical new inference rule may turn to be unsound. The (in-) famous example of this is the Substitution Law, which originally was added as an axiom in classical UNITY [4]. The axiom turned to be unsound, as shown in 1990 by Sanders [19]. It takes several more years for people to realize that the correction given in the same paper [19] is also not without flaw [15].

In 1992, Andersen successfully mechanized and verified the inference rules of UNITY within the theorem prover HOL [1]. This was a milestone, as it showed a feasible and reliable way to check the soundness of UNITY like logics. Later, Prasetya mechanized some variants of UNITY in HOL, including a variant that removes the flaw in Sanders UNITY [13]. This turned out to be quite a time consuming task. To verify a new UNITY variant, Andersen's HOL code only served as, at best, guidance. One basically had to redo the entire soundness proof for each UNITY variant. This is where the idea and necessity for a tool like $\forall$UNITYwas born.

$\forall$UNITYis a generalization of UNITY. It provides the same set of inference rules, but these are now derived from a set of more primitive (weaker) rules. Using $\forall$UNITYa UNITY variant can be easily created by only showing that the variant upholds $\forall$UNITYprimitive rules. This is significantly less work than having to redo the entire UNITY soundness proof. $\forall$UNITYis provided as a HOL (a theorem prover) theory. All its derived laws have been mechanically verified. HOL ensures that any concrete UNITY variant which is derived from $\forall$UNITYis sound and complete, in the sense that it will satisfy all standard UNITY inference rules.

Effort has been made to make the derived rules of $\forall$UNITYminimal in the sense that each rule explicitly mentions which of the $\forall$UNITYprimitive rules it requires. Consequently, it is possible to make an instantiation that, for example, only upholds the Completion Rule under certain circumstances.

$\forall$UNITYis also useful for constructing general theories: any application theory which is based purely on $\forall$UNITYis also valid for all UNITY variants which can be instantiated from $\forall$UNITY. $\forall$UNITYcan be downloaded from:

`www.cs.uu.nl/~wishnu/research/research.html`.

1.1 Contents of the paper

Section 2 gives a brief review on classical UNITY. Section 3 explains the notation we use in this paper. Some examples of instances of $\forall$UNITYare described in Section 4. Section 5 presents $\forall$UNITY. Finally, Section 6 concludes.

Section 5 is somewhat informal. The formal definition of ∀UNITY's primitive properties is listed in Appendix A. The most important derived inference rules of ∀UNITYare listed in Appendix B. More derived laws can be found in the distribution package of ∀UNITY. Appendix C briefly comments on the relation with New UNITY.

2 Brief Review: UNITY

In UNITY, a distributed system is modeled by a set of actions, each of which is assumed to be atomic and terminating. An (concurrent) execution of such a system is an infinite and interleaved execution of its actions. In each step of the execution some action is non-deterministically selected from the set of enabled actions. Finite stuttering (skip) is allowed in the executions. The selection of actions is assumed to be weakly fair, meaning that an action which is continually enabled (waiting to be executed) can not be ignored forever.

To specify the behavior of a program, three operators are provided, namely unless, ensures, and $\mapsto$. Given two state predicates p and q, a program P is said to satisfy p unless q if: once p holds during an execution of P, it remains to hold at least until q holds. The program satisfies p ensures q if it satisfies p unless q and moreover there exists an action in P that can, and because of the fairness assumption of UNITY, will establish q. Their formal definitions are as follows:

Definition 2.1 : Unless and Ensures (Classical)

$$P \vdash p \ \mathsf{unless}\ q \ = \ (\forall a : a \in P : \{p \wedge \neg q\}\ a\ \{p \vee q\})$$

$$P \vdash p \ \mathsf{ensures}\ q \ = \ (P \vdash p \ \mathsf{unless}\ q) \ \wedge \ (\exists a : a \in P : \{p \wedge \neg q\}\ a\ \{q\})$$

□

Whereas unless specifies safety, ensures specifies progress. However, an ensures property can only specify progress which can be guaranteed by a single action. To describe progress in general, we use $\mapsto$. Informally, a program P satisfies $p \mapsto q$ if: whenever p holds during the execution of P then eventually q will also hold. Formally, $\mapsto$ is defined as the smallest transitive and left-disjunctive (i.e. at the p-position) closure of ensures.

To reason about the behavior of a program, the UNITY logic provides a set of inference rules. For example, one of the rules says that we can join two unless properties:

Theorem 2.2 : Unless General Conjunction (Classical)

$$\frac{\begin{array}{l} P \vdash p \ \mathsf{unless}\ q \\ P \vdash r \ \mathsf{unless}\ s \end{array}}{P \vdash p \wedge r \ \mathsf{unless}\ (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

□

As another example, the following rule states that if a program P can progress from p to q, and it can maintain a condition a at least until b holds, then starting from $p \wedge a$ it can either reach q while a still holds, or else then at least we know that it has entered the condition b. The rule is known as the *Progress Safety Progress* or PSP rule:

Theorem 2.3 : PSP (Classical)

$$\frac{\begin{array}{l} P \vdash p \mapsto q \\ P \vdash a \ \mathsf{unless}\ b \end{array}}{P \vdash (p \wedge a) \mapsto (q \wedge a) \vee b}$$

□

For the complete list of UNITY rules see for example [4]. The rules (in the case of classical UNITY: without the Substitution law) are sound and in fact can be derived from the definitions of the three UNITY operators (unless, ensures, and $\mapsto$).

3 Notation

We will use a notation that deviates from the usual UNITY style such as the one used in Section 2. Like UNITY, ∀UNITYis a formalism. However, it is implemented in the theorem prover HOL [7] as a HOL theory. Consequently, we will use the *HOL notation*, which is admittedly less stylish, but will make it easier for the reader to access the ∀UNITYHOL library.

Formulas are written in the type writer font and UNITY operators are written in the prefix-style, e.g. `UNLESS p q`. A UNITY inference rule will be written like this:

```
|- A1 /\ ... /\ An ==> C
```

which means that `C` is derivable from `A1` ... `An`. The notation makes an inference rule looks like an ordinary predicate logic theorem, which in fact it is: in ∀UNITYan inference rule is implemented as a HOL theorem (this also means that it is only a rule if its validity can be proven).

When defining a concrete UNITY (i.e. a ∀UNITYinstance), we will write the UNITY operators with upper case letters, e.g. `UNLESS`, `ENSURES`, and `LEADSTO`. These upper case names refer to concretely defined objects. In ∀UNITYitself these operators are parameters and we will write them with lower case letters, e.g. `unless`, `ensures`, and `leadsto`.

There are two levels of logical operators in ∀UNITY. We have:

`/\`	(conjunction)
`\/`	(disjunction)
`==>`	(implication)
`~`	(negation)
`!`	(universal quantification)
`?`	(existential quantification)

with the usual meaning. Semantically, in HOL they are boolean operators. For example, `/\` takes two booleans and returns a boolean.

A state-predicate is a predicate like $x > y+1$ and is used to, for example, specify the set of possible states a program can be in at a given moment. The predicate can be simply represented by `x>y+1` in HOL. However, doing so will prevent us from properly representing UNITY inference rules in HOL. The reason is rather technical, see for example [3]. The standard way to get around this is to represent a state-predicate semantically in HOL as a function from some type `'s` representing the universe of state to the type `bool` [1, 16, 21, 3]. So, we have another set of operators:

`AND`, `OR`, `IMP`, `NOT`, `!!`, and `??` which are just the previously listed boolean operators lifted to the state predicate level. For example, `AND` is defined as:

```
p AND q  =  (\s. p s /\ q s)
```

Abstractly though, the reader can pretend that both sets of operators are equivalent.

If `p` is a predicate (over some type `'s`), `valid p` is defined as follows:

Definition 3.1 : Valid Predicate

```
|- !p. valid p = !s. p s
```

□

Given some type `'s` representing the universe of states, an action is modelled as a relation over `'s`. So it is a function of type `'s->'s->bool`. Hoare triples are defined as follows:

Definition 3.2 : Hoare Triple

```
|- !p a q. HOA p a q = !s t. p s /\ a s t ==> q t
```

□

4 Examples of UNITY variants

The definition of ∀UNITYis given in Section 5, in this section we first take a look at three concrete examples of UNITY variants which can be (quite easily) obtained from ∀UNITY. The first example is classical UNITY which we have seen in the previous section –it will now be presented in the ∀UNITYnotation. The second example is Sanders' UNITY [19] which extends classical UNITY with a new ability. The final example is a UNITY variant called COMUNITY, proposed by Prasetya, Vos, Swierstra, and Widjaja [18]. It boosts UNITY even further using a relatively simple extension.

4.1 GLEADSTO

When instantiating ∀UNITY, typically we define the `LEADSTO` operator in the same way as in classical UNITY, namely as the smallest transitive and left-disjunctive closure of the `ENSURES` relation. However, the used `ENSURES` may be different from the classical one. Given a binary relation `U`, in ∀UNITYthe smallest transitive and left-disjunctive closure of `U` is written as `GLEADSTO U` –see Appendix A for a formal definition.

4.2 Classical UNITY

Here is how classical UNITY is formulated in the ∀UNITYnotation. A program is represented abstractly as a set of actions. A set of items of type `'t` is here represented by a function (a predicate) of type `'t->bool`.

Definition 4.1 :

1.
   ```
   |- !P p q.
        UNLESS P p q
          =
          (!a. P a ==> HOA (p AND NOT q) a (p OR q))
   ```

2.
   ```
   |- !P p q.
        ENSURES P p q
          =
          UNLESS P p q /\
          (?a. P a ==> HOA (p AND NOT q) a q)` ;
   ```

3. `|- !P. LEADSTO P = GLEADSTO (ENSURES P)`

□

4.3 Sanders' UNITY

In classical UNITY we cannot use our implicit knowledge about the invariant of a program to simplify a given UNITY specification. In [19] Sanders offers a simple solution to this, namely by extending all UNITY operators with a new parameter. With this parameter the user can specify an invariant, which subsequently can be used to simplify the other parameters.

Here a program `P` will be represented by a pair `(A,init)` where `A` represents the program's set of actions and `init` is a predicate specifying the program's initial condition. `ACTIONS P` and `INIT P` will return `A` and `init` respectively. Invariant is defined as follow (note that the definition is weaker that the one originally used in [19], which is unsound [15]):

Definition 4.2 : Invariant

```
|- !P J.
   INV P J
     =
     valid((INIT P) IMP J) /\ (!a. ACTIONS P a ==> HOA J a J)
```

□

Here is the concrete definition of Sanders' UNITY.

Definition 4.3 :

1.
```
|- !P J p q.
     UNLESS P J p q
       =
       INV P J /\
       (!a. ACTIONS P a ==> HOA (J AND p AND NOT q) a (p OR q))
```

2.
```
|- !P J p q.
     ENSURES P J p q
       =
       UNLESS P J p q /\
       (?a. ACTIONS P a ==> HOA (J AND p AND NOT q) a q)
```

3. `|- !P J. LEADSTO P J = GLEADSTO (ENSURES P J)`

□

4.4 COMUNITY

COMUNITY (COMpositional UNITY) [18] increases the power of Sanders UNITY even further:

1. COMUNITY only requires the `J` parameter to be an stable rather than invariant. A predicate `J` is stable in a program `P` if `P` cannot destroy it:

   Definition 4.4 : Stable Predicate

   ```
   !P J. STABLE P J  =  !a. P a ==> HOA J a J
   ```

□

Evidently (see Definition4.2), a stable predicate that also holds initially is an invariant. By weakening the requirement of invariance of `J` to stability of `J`, it is possible in COMUNITY to specify properties of a program which may only be reachable when the program is executed in parallel with other programs.

2. COMUNITY allows the user to specify the sensitivity of a property, say `X`, of a program `P` to external interference. This is specified in an additional parameter `A` which is a set of predicates which are indestructible by the environment. Obviously, the property `X` is preserved when `P` is composed with an environment that maintains (each predicate in) `A`. COMUNITY also comes with a set of (new) proof rules to, in the less trivial case, compose a program with an environment which can only maintain `A` temporarily[2].

Definition 4.5 :

1.
```
|- !P J A p q.
     UNLESS P J A p q
       =
       STABLE P J /\ A p /\ A q /\
       (!a. P a ==> HOA (J AND p AND NOT q) a (p OR q))
```

2.
```
|- !P J A p q.
     ENSURES P J A p q
       =
       UNLESS P J A p q /\
       (?a. P a ==> HOA (J AND p AND NOT q) a q)
```

3. `|- !P J A. LEADSTO P J A = GLEADSTO (ENSURES P J A)`

□

5 ∀UNITY

∀UNITYis the general version of classical UNITY. It is general because it does not impose any concrete interpretation of UNITY operators. Instead, it gives a set of quite weak primitive inference rules (primitive properties) that abstractly model a minimal requirement to obtain a UNITY-like logic. We have proven that when all the primitive properties of ∀UNITYare satisfied, then all standard UNITY inference rules are valid. An arbitrary UNITY variant can be created by providing a concrete definition of the relation 'unless' and 'ensures' and then showing that they satisfy ∀UNITYprimitive rules. We will refer to this process as *instantiating* ∀UNITY, and the resulting concrete UNITY logic will also be referred to as a UNITY *variant* or an *instance* of ∀UNITY.

For each derived inference rule in ∀UNITY, we specify the minimal set of primitive properties that have to be satisfied in order for the inference rule to be valid. So, a user can create an instance where some primitive properties are not satisfied, or only conditionally satisfied. In the first case, the created instance simply inherits less derived rules. In the second case, for example when a primitive property R only holds under a certain condition C, we can always propagate the condition C to all derived rules that depend on R.

[2]For example, suppose `A` can be maintained by the environment `Q` during a time interval which is characterized by the state predicate `a`; suppose the program `P` can do the progress `LEADSTO P J A p q`. We can infer that the composition `P PAR Q` will satisfy: `LEADSTO (P PAR Q) J A (p AND a) (q OR NOT a)`. See [18] for further reading.

All three UNITY variants described in Section 4 specify a program property using an expression of the form `UOP V1 ... Vn p q` where: `UOP` is a UNITY operator (`UNLESS`, `ENSURES`, or `LEADSTO`); `V1` ... `Vn` are what we will refer to as *extensional parameters*; and `p` and `q`, from now on called the `pq`-*parameters*, are those parameters that all UNITY variant agree on (i.e. in all variants `LEADSTO ... p q` specifies a property in which if `p` holds then eventually `q` will also hold). For example, a classical UNITY property mainly depends on the `pq`-parameters, having only one extensional parameter that models the program[3] `P` the property is related to. The other UNITY variants have plugged in new functionality by simply adding more extensional parameters and defining their relation with the `pq`-parameters. Consequently, this form of UNITY properties is very general, and all UNITY variants whose operators can be written in this form can be instantiated from ∀UNITY.

∀UNITYonly focuses on the `pq`-parameters, because the interpretation of the extensional parameters and their relation with the `pq`-parameters are variant specific and hence beyond the scope of ∀UNITY. However, we cannot completely abstract away from the extensional parameters, since, whatever they are, the information in the `pq`-parameters is some way related to these extensional parameters. To capture this relation between the `pq`-parameters and the extensional parameters, ∀UNITYhas a new operator called `implies`, which, as the name suggests, behaves in many ways like the ordinary $\Rightarrow$ operator. The characterisation of `implies` is in the next Subsection 5.1). Furthermore, Subsection 5.7 provides a number of theorems capturing some general forms of inference rules concerning extensional parameters.

In the three UNITY variants described in Section 4, the UNITY operators are concretely defined, that is `UNLESS`, `ENSURES` and `LEADSTO` are (HOL) constants. In ∀UNITY, however, they are not concretely defined, and in all inference rules of ∀UNITYthey are universally quantified variables. As indicated before, to emphasis this distinction, in ∀UNITYthe operators are written like lower case characters, e.g. `unless` and `ensures`. Note that these variables model binary operators: they only have the `p` and the `q` parameter.

5.1 The Implies Operator

As said, ∀UNITYallows one more operator to be specified, namely `implies`. It is used to specify the fact that in some given temporal situation a state predicate `p` implies another state predicate `q`. This is mainly used to simplify a specification. For example, suppose a program has the property `leadsto p q`. If we also have `implies q r` and `implies r q` then we know that in 'this situation' `q` and `r` are equivalent, and hence `q` can be simplified to `r`. The situation under which the implication holds is typically specified in the extensional parameters. For example, in the Sanders' UNITY this is specified in the `J` parameter:

Definition 5.1 : Sanders' Implies

```
|- !J p q. IMPLIES J p q = valid (J AND p IMP q)
```

□

In Sanders' UNITY `J` is intended to be an invariant of a program. The above concrete definition of `implies` means that we are allowed us to use what we know about the program's invariants to infer the implication, and ultimately, to simplify the `pq`-parameters of the UNITY specifications of the program.

[3]The fact that the program is an extensional parameter, means that in ∀UNITYthere is no explicit requirement that operators actually have anything to do with programs.

In the classical UNITY, we do not have the `J` parameter and `implies` corresponds to the usual predicate logic `IMP`:

Definition 5.2 : Classical Implies

```
|- !p q. IMPLIES p q = valid (p IMP q)
```

□

COMUNITY has a more restricted notion of `implies`, which allows us to infer implication only when certain properties of the environment are also satisfied:

Definition 5.3 : COMUNITY Implies

```
|- !J A p q.  IMPLIES J A p q = A p /\ A q /\ valid (J AND p IMP q)
```

□

5.2 Domain of the Operators

In the classical UNITY the `p` and `q` in, for example, `UNLESS p q` can be any predicates. So, the *domain* of the operator is simply the entire universe of predicates. However, in other UNITY variants this may not be the case. For example, above we have seen that in COMUNITY the domain of the `pq`-parameters of the `IMPLIES` operator is also restricted by `J` and `A`. For later, it is useful to introduce the notation `inDomain U` to denote the domain of a UNITY operator `U`. There are a number of ways to characterize `inDomain`; we choose the following. Observe that 'reflexiveness' is a desired (natural) property for any UNITY operator. That is, for any `p` in the domain of a given UNITY operator `U`, we want `U p p` to be a valid property –but only, for `p`'s which are in the domain of `U`. So we can just as well characterize `inDomain` as follows:

Definition 5.4 : In Domain

```
    inDomain U p = U p p
```

□

In general, this is not a complete characterization of 'domain', however for UNITY operators it is.

5.3 ∀UNITYPrimitive Rules

Below we list the primitive inference rules (properties) of ∀UNITY. The properties will be divided into three groups: the I, D, and N groups. Some informal explanation will be provided. Their formal definition is listed in Appendix A.

5.3.1 I Properties

Abstractly, the `implies` operator behaves just like the ordinary ⇒ operator. They are however not equivalent. In a given UNITY variant `implies` may use information in the extensional parameters to conclude the implication. The following properties specify how much of the ⇒-behavior we need in `implies`.

1. `isClosed_under_PredOPS (inDomain implies)`

   This says that `implies` should be closed under the standard predicate logic operators.

2. `includesIMP implies`

   This requirement says that if `p ==> q` holds, then `implies p q` should also hold, but only if both `p` and `q` are in the 'domain' of `implies` –see also the next subsubsection.

5.3.2 D Properties

These are domain constraining properties. A typical domain requirement in ∀UNITYis the one that says, for example, that if `unless p q` holds then both `p` and `q` must be in the domain of `unless`. This is written `hasProperDomain unless` in ∀UNITY. Another example is the requirement that says that the domain of `unless` should be included in the domain of `implies`:

```
!p. inDomain unless p ==> inDomain implies p
```

Since `implies` provides the link to the extensional parameters, it is this kind of requirement that ensures that the `pq`-parameters are always consistent with the extensional parameters. So, if we recall the COMUNITY example at the beginning of this subsubsection, the above requirement ensures that in `UNLESS P J A p q`, both `p` and `q` are members of `A`.

Here are the D properties:

1. `hasProperDomain implies`
2. `hasProperDomain unless`
3. `!p. inDomain unless p ==> inDomain implies p`
4. `hasProperDomain ensures`
5. `!p. inDomain ensures p ==> inDomain implies p`

5.3.3 N Properties

In all three UNITY variants from the previous section, `unless` and `ensures` are defined in terms of the next-state behavior of the specified program. However, the next-state behavior of the program may depend on the situation derivable only from the information in the extensional parameters of `unless`. This information is abstracted away in ∀UNITY, so obviously we cannot in ∀UNITYdefine the operators in the same way. Fortunately, there is another way. The following primitive properties abstractly characterize the intended temporal properties described by ∀UNITYoperators:

1. `isSubRelationOf implies unless`

   This says that for all `p` and `q`, if `implies p q` then `unless p q`.

2. `satisfiesUNLESS_AntiRefl unless`

   This says that `unless` is anti-reflexive. So, `unless p (NOT p)` is always a valid property, provided `p` is in the domain of `unless`.

3. `satisfiesUNLESS_Conj unless`

   This says that `unless` satisfies the General Conjunction rule –see also Theorem 2.2.

4. `satisfiesUNLESS_Disj unless`

   This says that `unless` satisfies the General Disjunction rule [4], which is the dual of the the General Conjunction rule.

5. `satisfiesUNLESS_Subst implies unless`

   This says that `unless` satisfies the Substitution rule [4]. More precisely, if the following hold:

```
implies p q /\ implies q p /\ implies r s
```

Then we can replace `unless q r` with `unless p s`. Notice that the condition for substitution is expressed in terms of `implies` whose concrete definition is left unspecified in ∀UNITY. As remarked earlier, in Sanders' UNITY `implies` can be expected to carry information about a program's invariant. In the classical UNITY `implies` is simply `IMP`. The substitution rule as formulated above is still satisfied, though it then becomes much less powerful than the one obtained in Sanders' UNITY.

6. `isSubRelationOf implies ensures`

7. `satisfiesPSP ensures unless`

   This says that `ensures` also satisfies the Progress Safety Progress (PSP) rule [4]. This property is necessary for deriving the PSP rule for `leadsto`.

8. `isSubRelationOf ensures unless`

   This says that `ensures` is a more restricted form of `unless`. Most UNITY variants take this property for granted: `ensures` is just `unless` strengthened with a requirement on the existence of some helpful action/transition. Curiously, the only derived inference rule that depends on this property is the Completion rule.

9. `leadsto = GLEADSTO ensures`

   This says that `leadsto` has to be defined as the least transitive and left-disjunctive closure of `ensures`.

5.4 Instantiating ∀UNITY

As an example, to instantiate ∀UNITYto COMUNITY, we substitute:

`implies`	with	`IMPLIES P J A`
`unless`	with	`UNLESS P J A`
`ensures`	with	`ENSURES P J A`
`leadsto`	with	`LEADSTO P J A`

where `IMPLIES` is defined in Definition 5.3; the other upper case operators are defined in Subsection 4.4. We should also substitute `GLEADSTO ensures` in ∀UNITYrules with `LEADSTO P J A`.

5.5 Derived Inference Rules

To give some idea, we will show some of the derived inference rules of ∀UNITY. The distribution package itself contains much more rules. The most important of them are listed in Appendix B.

The following is ∀UNITY's version of the general conjunction rule for `unless`.

Theorem 5.5 : Unless Simple Conjunction Rule

```
|- includesIMP implies                      /\
   isClosed_under_PredOPS (inDomain implies) /\
   (!p. inDomain unless p ==> inDomain implies p) /\
   hasProperDomain unless                   /\
   satisfiesUNLESS_Subst implies unless     /\
   satisfiesUNLESS_Conj unless              /\
   unless p1 q1                             /\
   unless p2 q2
   ==>
   unless (p1 AND p2) (q1 OR q2)
```

□

The first six assumptions specify the ∀UNITY's primitive properties required to derive the familiar form of the simple conjunction rule as formulated in the classical UNITY.

The following inference rule is ∀UNITY's version of the PSP rule. Compare it with its usual form in Theorem 2.3.

Theorem 5.6 : PSP

```
|- includesIMP implies                             /\
   isClosed_under_PredOPS (inDomain implies)       /\
   hasProperDomain unless                          /\
   (!p. inDomain unless p ==> inDomain implies p)  /\
   hasProperDomain ensures                         /\
   isSubRelationOf implies ensures                 /\
   (!p. inDomain ensures p ==> inDomain implies p)  /\
   satisfiesPSP ensures unless                     /\
   GLEADSTO ensures p q                            /\
   unless a b
   ==>
   GLEADSTO ensures (p AND a) (q AND a OR b)
```

□

5.6 The Completion Rule

Progress is generally disjunctive but not generally conjunctive. For example, if a program P can progress from p to q and from p to r we do not know if it can progress to a state where both q and r hold. However, if we know that, for example, both q and r are stable predicates (Definition 4.4), then we know that the conjunction of them will hold eventually. This kind of property of progress is often very useful, e.g. as in [9, 17]. In UNITY, this is captured by the so-called Completion Rule. Before we present ∀UNITY's version of the rule, let us first give a number of (new) derived operators to abstractly represent the kind of progress which is conjunctive:

Definition 5.7 : Completion

```
|- !progress unless p q b.
    COMPLETES leadsto unless p q b = leadsto p q /\ unless q b
```

□

Definition 5.8 : Convergence

```
|- !leadsto unless p q.
    CONVERGES leadsto unless p q = COMPLETES leadsto unless p q FF
```

□

For a given `leadsto` and `unless` relations, `COMPLETES leadsto unless p q b` models the progress from `p` to `q` by a program, and moreover, once `q` is reached the program will either remain in `q` or decide to exit to `b`. Obviously, if `b` is `FF` then the program cannot exit from `q` and will thus remain in `q`. The latter situation is modelled by the `CONVERGES` operator.

The Completion Rule looks like this in ∀UNITY:

Theorem 5.9 : Completion Rule

```
|- includesIMP implies                            /\
   (!p. inDomain unless p ==> inDomain implies p)  /\
   isClosed_under_PredOPS (inDomain implies)      /\
   hasProperDomain unless                         /\
   isSubRelationOf implies unless                 /\
   isSubRelationOf implies ensures                /\
   satisfiesUNLESS_Subst implies unless           /\
   satisfiesUNLESS_Conj unless                    /\
   satisfiesUNLESS_Disj unless                    /\
   satisfiesPSP ensures unless                    /\
   hasProperDomain ensures                        /\
   (!p. inDomain ensures p ==> inDomain implies p)  /\
   isSubRelationOf ensures unless                 /\
   satisfiesPSP ensures unless                    /\
   COMPLETES (GLEADSTO ensures) unless p q b      /\
   COMPLETES (GLEADSTO ensures) unless r s b
   ==>
   COMPLETES (GLEADSTO ensures) unless (p AND r) (q AND s OR b) b
```

□

This rule is the most difficult to prove and the most demanding one, as it requires almost all ∀UNITYprimitive properties (the first 14 assumptions above). If these primitive properties can be discharged then we will get the standard Completion rule. As remarked earlier, this is the only rule where we actually use the fact that `ensures` is defined as a restricted form of `unless`. As a corollary we can easily show that convergence is conjunctive:

Corollary 5.10 : Convergence Conjunction

```
|- ...

   CONVERGES (GLEADSTO ensures) unless p q  /\
   CONVERGES (GLEADSTO ensures) unless r s
   ==>
   CONVERGES (GLEADSTO ensures) unless (p AND r) (q AND s)
```

where `...` stands for the same set of conditions as in Theorem 5.9.
□

5.7 Inferring and Composing Extensional Parameters

In the beginning of this section we have said that ∀UNITYis intended to focus on the `pq`-parameters. Consider a COMUNITY specification `LEADSTO P J A p q`. Recall that the `p` and `q` are called the `pq`-parameters. The rest are called extensional parameters. The `implies` operator can be used to capture the relation between the two kind of parameters, but beyond that ∀UNITYbasically do not provide any support to reason about the extensional parameters (because that kind of reasoning is too variant specific). However, we notice that the following two kinds of inference capabilities are often desired. We will illustrate them with COMUNITY examples:

1. `LEADSTO P J A p q` implicitly implies certain properties of the extensional parameters. For example, it implies that `J` is stable in `P`.

2. Changing the value of the extensional parameters typically destroy a property, however, if we only change them in a certain way, the property may be preserved after all. For example, `LEADSTO P J A p q` is preserved if we strengthen `J` with another stable predicate. The property is also preserved if we compose `P` and `A` with another program and another set of predicates provided they satisfy certain constraints [18]. Inference rules for program composition, e.g. the Union rules [4] and Singh rules [20, 11, 18], fall into this category.

∀UNITYprovides two theorems to accommodate those kinds of inference, but will not attempt to characterize the specific conditions under which they are applicable since this is something which is too variant specific. We introduce first the following two operators:

Definition 5.11 : Implicitly Implies

```
|- !unityOp property.
    implicitlyImplies unityOp property
    =
    !A p q. unityOp A p q ==> property A
```

□

So, for example:

```
     implicitlyImplies (\J LEADSTO P J A) (STABLE P)
```

captures what we said earlier, namely that in COMUNITY `LEADSTO P J A p q` implies the stability of `J P`.

Definition 5.12 : Conservative Extension

```
|- !unityOp extend.
    isConservative unityOp extend
    =
    !A p q. unityOp A p q ==> unityOp (extend A) p q
```

□

So, for example:

```
     isConservative (\J. LEADSTO P J A) (\J. J AND J')
```

states that in COMUNITY the property `LEADSTO P J A p q` is preserved when we strengthen `J` to `J AND J'` (for some fix `J'`).

It is then quite trivial to obtain the following theorems:

Theorem 5.13 : Implicitly Implies Rule

```
|- implicitlyImplies unityOp property /\
   unityOp A p q
    ==>
   property A
```

□

Theorem 5.14 : Conservative Extension Rule

```
|- isConservative unityOp extend  /\
   unityOp A p q
   ==>
   unityOp (extend A) p q
```

□

When `unityOp` is leads-to then we also have the following stronger theorems:

Theorem 5.15 :

```
|- implicitlyImplies ensures_ property /\
   GLEADSTO (ensures_ A) p q
    ==>
   property A
```

□

Theorem 5.16 :

```
|- isConservative ensures_ extend  /\
   GLEADSTO (ensures_ A) p q
   ==>
   GLEADSTO (ensures_ (extend A)) p q
```

□

6 Conclusion

∀UNITYis a generalization of UNITY. It provides the same set of inference rules as the standard UNITY but it does not impose any concrete interpretation of what `unless` and `ensures` are. Instead, it gives a set of quite weak primitive properties that abstractly model a minimalistic requirement to obtain a UNITY-like logic. A user can create an arbitrary variant of UNITY by providing a concrete definition of `unless` and `ensures` and then showing that they satisfy ∀UNITYprimitive rules.

Furthermore, for each derived inference rule in ∀UNITY, we specify which primitive properties it minimally requires. Hence, it is possible to create a weaker ∀UNITYinstance where not all primitive properties are satisfied, or only conditionally satisfied.

∀UNITYis provided as a HOL (a theorem prover) library. Creating a UNITY variant using ∀UNITYhas the following advantages:

1. A rich set of standard inference rules have already proven; this saves a lot of work.

2. The rules have been proven *mechanically* in HOL, so they are very safe to use.

3. The user automatically gets all theorem proving support of HOL.

∀UNITYis suitable for creating a UNITY variant where properties are specified like this: `UnityOperator V p q` where `p` and `q` are state predicates having the usual UNITY meaning and `V` is a list of additional parameters. This is a very general form. In the classical UNITY we only have one additional parameter to specify the program to which the specified property belong. In more sophisticated variants, e.g. Collete and Knapp's variants [6, 5], closures operators in the new UNITY [12], and COMUNITY [18], more parameters are added in order to attach more functionalities to the standard UNITY.

∀UNITYis also useful for constructing general theories: an application theory which is based purely on ∀UNITYwill also be valid for all UNITY variants which can be instantiated from ∀UNITY.

A Definition of ∀UNITYPrimitive Properties

1.
   ```
   |- !dom.  isClosed_under_PredOPS dom = dom TT /\ (!p. dom p ==> dom
   (NOT p)) /\ (!p q. dom p /\ dom q ==> dom (p AND q)) /\ (!W. (!p. W p
   ==> dom p) ==> dom (!!p::W. p))
   ```

2.
   ```
   |- !U.
      includesIMP U
        =
        !p q. inDomain U p /\ inDomain U q /\ valid (p IMP q) ==> U p q
   ```

3.
   ```
   |- !U.
      hasProperDomain U
        =
        !p q. U p q ==> inDomain U p /\ inDomain U q
   ```

4.
   ```
   |- !U p. inDomain U p = U p p
   ```

5.
   ```
   |- !U V. isSubRelationOf U V = !x y. U x y ==> V x y
   ```

6.
   ```
   |- !unless.
      satisfiesUNLESS_AntiRefl unless
        =
        !p. inDomain unless p ==> unless p (NOT p)
   ```

7.
   ```
   |- !unless.
      satisfiesUNLESS_Conj unless
        =
        !p1 q1 p2 q2.
         unless p1 q1 /\ unless p2 q2
         ==>
         unless (p1 AND p2) (q1 AND p2 OR q2 AND p1 OR q1 AND q2)
   ```

8.
   ```
   |- !unless.
      satisfiesUNLESS_Disj unless
        =
        !p1 q1 p2 q2.
         unless p1 q1 /\ unless p2 q2
         ==>
         unless (p1 OR p2) (q1 AND NOT p2 OR q2 AND NOT p1 OR q1 AND q2)
   ```

9.
   ```
   |- !implies unless.
      satisfiesUNLESS_Subst implies unless
        =
        !p q a b.
         unless p q /\ implies p a /\ implies a p /\ implies q b
         ==>
         unless a b
   ```

10.
    ```
    |- !progress unless.
       satisfiesPSP progress unless
         =
         !p q a b.
          progress p q /\ unless a b
          ==>
          progress (p AND a) (q AND a OR b)
    ```

11.
    ```
    |- !ensures p q.
       GLEADSTO ensures p q =
         !U. isSubRelationOf ensures U /\
             isTransitive U /\
    ```

```
      isLeftDisj U
      ==>
     U p q
```

12. `|- !leadsto.`

```
       isLeftDisj leadsto
       =
       !W q. (?p. W p) /\
            (!p. W p ==> leadsto p q)
            ==>
            leadsto (??p::W. p) q
```

B General UNITY Laws

1. `unless` reflexifity:

   ```
   |- inDomain unless p ==> unless p p
   ```

2. Lifting `implies` to `unless`:

   ```
   |- includesIMP implies /\ isSubRelationOf implies unless /\
      inDomain implies p  /\ inDomain implies q /\
      valid (p IMP q)
      ==>
      unless p q
   ```

3. Weakening the post-condition of `unless`:

   ```
   |- includesIMP implies /\
      (!p. inDomain unless p ==> inDomain implies p) /\
      hasProperDomain unless /\
      satisfiesUNLESS_Subst implies unless /\
      unless p q /\ implies q r
      ==>
      unless p r
   ```

4. Simple conjunctivity of `unless`:

   ```
   |- includesIMP implies /\
      isClosed_under_PredOPS (inDomain implies) /\
      (!p. inDomain unless p ==> inDomain implies p) /\
      hasProperDomain unless /\
      satisfiesUNLESS_Subst implies unless /\
      satisfiesUNLESS_Conj unless /\
      unless p1 q1 /\ unless p2 q2
      ==>
      unless (p1 AND p2) (q1 OR q2)
   ```

5. Simple disjunctivity of `unless`:

   ```
   |- includesIMP implies /\
      isClosed_under_PredOPS (inDomain implies) /\
      (!p. inDomain unless p ==> inDomain implies p) /\
      hasProperDomain unless /\ satisfiesUNLESS_Subst implies unless /\
         satisfiesUNLESS_Disj unless /\ unless p1 q1 /\ unless p2 q2 ==>
         unless (p1 OR p2) (q1 OR q2)
   ```

6. Lifting `implies` to 'leadsto':

```
|- includesIMP implies /\ isSubRelationOf implies ensures /\
   inDomain implies p /\ inDomain implies q /\
   valid (p IMP q)
   ==>
   GLEADSTO ensures p q
```

7. Lifting `ensures` to 'leadsto':

```
|- ensures p q ==> GLEADSTO ensures p q
```

8. Transitivity of 'leadsto':

```
|- GLEADSTO ensures p q /\ GLEADSTO ensures q r ==> GLEADSTO ensures p r
```

9. Left-disjunctivity of 'leadsto'[4]:

```
|- W i /\ (!i. W i ==> GLEADSTO ensures (f i) q)
   ==>
   GLEADSTO ensures (??i::W. f i) q
```

10. Simple disjunction for 'leadsto':

```
|- includesIMP implies /\
   isClosed_under_PredOPS (inDomain implies) /\
   isSubRelationOf implies ensures /\
   hasProperDomain ensures /\
   (!p. inDomain ensures p ==> inDomain implies p) /\
   GLEADSTO ensures p q /\ GLEADSTO ensures r s
   ==>
   GLEADSTO ensures (p OR r) (q OR s)
```

11. Substitution rule for 'leadsto':

```
|- isSubRelationOf implies ensures /\
    implies a p /\ implies q b /\
    GLEADSTO ensures p q
    ==>
    GLEADSTO ensures a b
```

12. Cancellation rule:

```
|- includesIMP implies /\
   isClosed_under_PredOPS (inDomain implies) /\
   isSubRelationOf implies ensures /\ hasProperDomain ensures /\
   (!p. inDomain ensures p ==> inDomain implies p) /\
   inDomain (GLEADSTO ensures) q /\
   GLEADSTO ensures p (q OR r) /\ GLEADSTO ensures r s
   ==>
   GLEADSTO ensures p (q OR s)
```

13. Bounded progress rule:

```
|- includesIMP implies /\
   isClosed_under_PredOPS (inDomain implies) /\
   isSubRelationOf implies ensures /\ hasProperDomain ensures /\
   (!p. inDomain ensures p ==> inDomain implies p) /\
   ADMIT_WF_INDUCTION LESS /\
```

[4]The `W i` condition is equivalent with `(?i. W i)`. Hence, we require `W` to be non-empty. If `W` is empty, then `(??i::W . f i)` is equivalent to `FF`. Although operationally progress can be made from `FF` to any `q`, unlike the classical UNITY, ∀UNITYonly allows `GLEADSTO ensures FF q` to be inferred if `q` is actually in the domain of `ensures`. See also the `implies` lifting rule. Keeping the expressions confined inside their domain is important to keep the `pq`-parameters consistently with the extensional parameters. See also the discussion in Subsubsection 5.3.2

```
inDomain (GLEADSTO ensures) q /\
(!m. GLEADSTO ensures
        (p AND (\s. M s = m))
        (p AND (\s. LESS (M s) m) OR q))
==>
GLEADSTO ensures p q
```

C Relation with New UNITY

In [12], Misra introduces New UNITY which is based on a set of new temporal operators: **co**, **transient**, **en**, and **leadsto**. The **leadsto** and **en** behave the same way as their classical counterparts, and **transient** is just an auxiliary operator used to define **en**. The operator **co** is an alternative to **unless**. It has nicer algebraic properties. On the other hand, **unless** has a more intuistic and familiar interpretation. The choice between them is probably a matter of taste. They can be defined in terms of each other. For example, here is how **co** can be defined in terms of **unless** in ∀UNITY:

Definition C.1 : CO

```
|- !P J A p q.
    CO implies unless p q = implies p q /\ unless p (NOT p AND q)
```

□

New UNITY also introduces the notion of *closure* which provides an abstraction for program composition. For example, the closure of the **leadsto** operator is called **cleadsto**, defined as follows:

$$P \vdash p \ \textbf{cleadsto}\ q \ = \ (\forall Q :: P \| Q \vdash p \ \textbf{leadsto}\ q)$$

Unlike in the classical UNITY, $P\|Q$ is only defined if Q satisfies P's link constraint. The link constraint of a program essentially specifies some upper bound on the kind of operations the environment can do on the variables of P.

We can also express new UNITY closured operators in terms of COMUNITY operators. So, $P \vdash p$ **cleadsto** q can be written as `LEADSTO P TT A p q` where `A` is suitably chosen to represent P's link constraint. Since COMUNITY is an instance of ∀UNITY–it satisfies all ∀UNITYprimitive rules–, so is the closured logic of New UNITY's closured operators.

References

[1] F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[2] A. Azurat and I.S.W.B. Prasetya. A preliminary report on xMECH. Technical Report UU-CS-2002-008, Inst. of Information and Comp. Science, Utrecht Univ., 2002. Download: `www.cs.uu.nl/staff/wishnu.html`.

[3] A. Azurat and I.S.W.B. Prasetya. A survey on embedding programming logics in a theorem prover. Technical Report UU-CS-2002-007, Inst. of Information and Comp. Science, Utrecht Univ., 2002. Download: `www.cs.uu.nl/staff/wishnu.html`.

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

[5] P. Collette and E. Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. *Lecture Notes in Computer Science*, 936:353–??, 1995.

[6] Pierre Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23(2–3):107–125, December 1994. Selected papers of the Colloquium on Formal Approaches of Software Engineering (Orsay, 1993).

[7] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[8] B. Heyd and P. Cregut. A modular coding of UNITY in COQ. *Lecture Notes in Computer Science*, 1125:251–266, 1996.

[9] P.J.A. Lentfert. *Distributed Hierarchical Algorithms*. PhD thesis, Utrecht University, April 1993.

[10] P.J.A. Lentfert and S.D. Swierstra. Towards the formal design of self-stabilizing distributed algorithms. *Lecture Notes of Computer Science*, 665:440–451, February 1993.

[11] J. Misra. Notes on UNITY 17-90: Preserving progress under program composition. Downloadable from `www.cs.utexas.edu/users/psp`, July 1990.

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

[13] I. S. W. B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Inst. of Information and Comp. Science, Utrecht Univ., 1995. Download: `www.cs.uu.nl/library/docs/theses.html`.

[14] I.S.W.B. Prasetya. Formalization of variables access constraints to support compositionality of liveness properties. *Lecture Notes in Computer Science*, 780:324–337, 1993.

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

[16] I.S.W.B. Prasetya. Formalizing unity with hol. Technical Report UU-CS-1996-01, Inst. of Information and Comp. Science, Utrecht Univ., 1996. Download: `www.cs.uu.nl/staff/wishnu.html`.

[17] I.S.W.B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. *Lecture Notes in Computer Science*, 1217:399 – 415, 1997.

[18] I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra, and B. Widjaja. A theory for composing distributed components based on mutual exclusion, 2002. Submitted for publication. Download: `www.cs.uu.nl/~wishnu`.

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

[20] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989.

[21] T. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms*. PhD thesis, Inst. of Information and Computer Sci., Utrecht University, 2000. Download: `www.cs.uu.nl`.

[22] T.E. Vos and S.D. Swierstra. Proving distributed hylomorphisms. Technical Report UU-CS-2001-40, Inst. of Information and Comp. Science, Utrecht University, 2001. Download: `www.cs.uu.nl`.