Towards Reliable Component Software: Light-weight Formalism

A. Azurat
Fakultas Ilmu Komputer, Universitas Indonesia, Indonesia
email: ade@cs.ui.ac.id

I.S.W.B. Prasetya
Inst. of Information and Computing Sciences, Utrecht University, Netherlands.
email: wishnu@cs.uu.nl

Abstract

The component software technology is a promising trend for rapid software development. However, one of the problems of software engineering is still inherited, which is the high cost of program verification. Moreover, not just the component, but also the composition of components is a subject for verification. In the direction of component software free-trade market, verification concerns become more important. A well defined formal foundation on component framework is required to solve it. We introduce a formal lightweight framework to analyse and verify the use of software component. We define some agreements on developing the component and its composition within a framework. By obeying those agreements, we argue that the verification cost, especially on progress property can be reduced. Some theorems that provide the judgement of those agreements are described. We first discuss how the underlying formalism is developed based on a variant of UNITY logic. This work reported in this extended abstract is part of an ongoing research. It contributes to a more reliable software product in the future.

Keywords: Component Based Development, light weight formal method, software specification validation and verification

1 Introduction

Software engineering has evolved from the object-oriented paradigm to component-based. In this new paradigm, a software vendor develops some components independently and composes them into one application. Software engineering in general still suffers from errors and bugs. It is still difficult to verify software formally. The verification cost can be higher than the development cost of the software. This verification problem is also inherited in component paradigm. It is even more difficult than in software engineering in general because in the component paradigm, not only the component has to be verified, but also the verification in the composition framework has to be preserved. The verification difficulty makes the secure-component become expensive or even impossible.

[10] mentioned that the component compositions cannot be verified formally without a well-defined foundation. There were some known logical formalizations ([5], [6], [4], [2]). One of the most related reports were provided by [2]. He represented the components and the compositions in a powerful logical formalization. However, the logic requires the component framework to generate many detailed verification conditions which are usually difficult to verify. Therefore, this formalization is difficult to apply. The work in this paper is although base on the same basic notion of logic with [6], but differ in the level of abstraction of the formalism. We provide another language layer than presented in [6].

We investigated the component framework with a lightweight formalization that encapsulates some of the formalization details to make it easier to apply. We allow a component to interact with its environment only through its method calls. This policy restricts the expressive power of developing and composing a component, but simplifies the composition verification and reduces the verification conditions.

We present an ongoing research on component framework based on component formalization in [9]. The formalization based on UNITY logic, which has been chosen because of its simplicity. The expressive power of the UNITY logic is less than the logic pro-

posed by Broy. However, it is sufficient to present the useful properties that need to be verified. Our framework uses PLP as the implementation language of the component. It is bassically an extention of Dijkstra's Guarded command language.

The article is organized as follows. Section 2 summarizes the UNITY logic and PLP language, Section 3 describes the architecture of the framework, Section 4 describes our light-weight formalism. Section 5 explain some proposed agreement on developing component software in this framework. The underlying concept for verification will be explain in Section 6. Finally, Section 7 concludes the article with some discussions.

2 UNITY and PLP Language

2.1 UNITY

Unity is a formalism introduced in [3]. It originally designed to reason about distributed system. It consists of a programming language to model programs, a specification language to express temporal properties, and a logic to prove them. We will briefly describe the variant of program and properties language that we use in this paper.

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

$Prog_{Unity} \triangleq (acts :: \{Action\}, init :: Pred, pub :: \{Var\}, pri :: \{Var\})$ $P.acts$ is a list of non-abortive action, which means every action should be terminated in an observed state result. $P.init$ captures possible initialisation state. $P.pub$ and $P.pri$ are respectively the public and private variable of the program. The notation $P.var$ is used for $P.pub \cup P.pri$. Because of some limitation of UNITY program such as no construct of sequence, we use it as the abstract model only, for the concrete program we use PLP.

A concrete program has to be the refinement of the abstract one.

Definition 2.1 *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:
$V, i \vdash P \sqsubseteq Q$
$\triangleq$
$P.pub \subseteq Q.pub \wedge P.pri \subseteq Q.pri \wedge Q.init \Rightarrow P.init$
$\wedge\ (\forall a : a \in Q.acts : V, i \vdash \sqcup P.acts \sqsubseteq a)$

Definition 2.1 says: under the invariance of $i, V, i \vdash P \sqsubseteq Q$ means that every action of Q behaves no worse than some action of P, with respect to the variable in V, or it just skips V. On the action, level we use the weak refinement, which allows skip. In this definition, we say that Q is a refinement of P, and P is an abstraction of Q.

The properties can be basically divided into safety and progress property. On safety property we use *stable* and *unless* operator. We use *ensures* and $\mapsto$ operator for progress property. Those operators require two predicate arguments, and are parameterised by a set of variables and a strong invariant.

We use *StrongInvariant* instead of just invariant to avoid the technical problem with substitution rule in UNITY as described in [8].

Definitions and basic theorems on these operators can be found in Appendix A.1 and A.2.

2.2 PLP

PLP stands for Programming language for Process. It is basically an extention of Dijkstra's Guarded Command Language. It is made in the style of UNITY program, but allows explicit sequencing. The execution model is the interleaving model of guarded action.

The grammar of PLP actions is as follows:

```
actions   ::= <action>
          |   <action> || <PLP actions>
          |   <action> ;  <PLP actions>
action    ::= <statement>
          |   g -> <statement>
statement ::= <variable> := <expression>
          |   <method call>
```

actions is a list of actions combined with construct $\|$ or ;, respectively for parallel and sequential composition. An *action* is either a statement of a guarded statement. A *statement* is either a simple assignment or a method call. We assume the action is conducted in *atomic*, that is to say, no interleaving during the method call nor variable assignment.

3 Framework Element

Before we explain the architecture of the framework, we define the elements of a framework, which are:

Component A component is a unit of deployment. It is a black-box implementation attached with a contract (identifies by name) and possibly the certification regarding the contract. A component should be independently deployed and independently verified. A component can have contract only with no implementation. It is called abstract component.

A component *assumes* and restricts its environment by the method call written in its contract. The interface to a component is its contract. The behaviour of a component is contrained by its properties and abstract code, which are defined in the contract. The variable state space of a component can be inspected directly but modifying is only allowed by calling its methods.

Our concrete definition of a component is : A software component is an encapsulated implementation, attached with a contract and possibly a certificate with respect to the contract.

The Semantic of a component is as followed :

Definition 3.1 *Component Semantic*

```
comp = ( impl     :: Implementation
       , contract :: Contract
       , cert     :: Certificate )
```

Contract A *contract* describes a component. It basically consists of:

- variable definitions,
- method definitions, as atomic action, confined with variables,
- functional properties written in UNITY logic,
- and abstract code.

A contract is the abstract specification and interface of a component. The semantic of a contract is defined as a tuple of:

Definition 3.2 *Contract Semantic*

```
Contract =  ( var  :: { Variable }
            , meth :: { Method }
            , prop :: { UnityProperty }
            , inv  :: { Predicate }
            , abs  :: { UnityProgram} )
```

The invariant is required to be parameterised with the property of the contract. The invariant may not represent the behaviour of a component, but it is useful and required in the formalism.

Client Script The *ClientScript* combines the components in a framework. A component cannot call methods of other components. All interaction between the components is conducted by the client script. The script should be simple so that its verification is minimal. The main verification task should only be in the verification of the component. The client script is a limited PLP, which is equivalence with Unity program. A client script can inspect the component state space, but cannot modify it. Modification only possible via available method calls.

Implementation An *implementation* is the concrete implementation of a component. The *implementation* can be certified with respect to the property of the contract. Its certification can be attached to the component. An *implementation* may have local variables and properties that implies the component properties. An *implementation* is a white-box. It consists of processes sharing the same variable space. There is no coordination language or combination script (glue code) at this level among the processes. The properties of the *implementation* can be verified with the complete code of its processes.

Process A Process is a PLP program which interpreted as having top level loop and explicit environment abstract code. A process is a white-box. Each process may have its own properties that can be used to implies the properties of *implementation*.

The process should have explicit environment abstract code. It will not reduce the verification task, but at least moving some verification tasks to process level. In process level the verification scope is smaller, therefor they are easier to prove. The explicit environment abstract code allows a process to be verified as independent as possible. Although the *process* deployment is not independent, after it created, it can later be used in other implementation instances, as long as the variable space is matched and the asbtract environment is refined.

Certification A certificate is a proof script. It should be machine readable, and can be checked by a proof checker.

Component certification proves that the implementation of a component is consistent with its contract properties.

The framework certification prove that framework properties can be derived from its components properties, and consistent with framework's scripts.

In this presentation we present a certificate as the UNITY logic proof script in higher order logic with additional judgement of the specific rule have.

Some of more advance issues of certification such as persistence, portability, automatic generation, run time certification check, non-functional certification are not covered here.

3.1 Architecture

A combination of components constructs a *framework* [1]. A framework contains some components, client script, properties and possibly the certificate of its properties. A framework can be encapsulated as a component. In theory, this encapsulation can continue recursively unlimited. However, usually one encapsulation level already gives enough abstract view of the component. Deeper encapsulation may hide some of the useful functionality of the components inside the framework.

We define $frm :: Framework$ as

Definition 3.3 *Framework Semantic*

```
frm =  (  comp       :: { Component }
       ,  props      :: { UnityProperty }
       ,  script     :: PLP )
```

In framework Fw we can extract:

Variable $Fw.var = \cup c : c \in Fw.comp.contracts : c.var$

General Environment $Fw.env = \| c : c \in Fw.comp : c.meth$ using $Fw.var$

We suggest that a framework is developed using a bottom up approach. The construction of a framework is derived from the availability of contracts and how they are combined.

The implementation of a component is developed in top-down approach. First, there should be a contract written, and then implement it. From an object oriented point of view, the contract serves as the abstract class of the component. The contract guides the implementation. It is possible that the implementation code is refined completely from the abstract code of the contract.

4 Formalism

This section describe the logical formalism of the framework. Starting with this section, we will use the following abbreviations:

[1] Usually a framework refer to an in complete implementation of the application, in this paper the existance of complete implementation is irrelevant.

$M \quad is \quad (\| c : c \in Fw.comp.contract : c.meth)$
$A \quad is \quad (\| c : c \in Fw.comp.contract : c.abs)$
$I \quad is \quad (\| i : i \in Fw.comp.impl : i)$
$E \quad is \quad Fw.env$
$S \quad is \quad Fw.scr$

$Fw.inv_A$, abstract invariant of the contract, is the conjuncted invariants of the component contract.

$Fw.inv_I$, concrete invariant of the implementation, is the conjucted invariants of the implementation of component.

4.1 Framework Refinement

Definition 4.1 *Semantic of Framework Refinement*

$inv \vdash A \sqsubseteq I$
$\triangleq$
$(\forall c : c \in Fw.comp : inv \vdash c.contr.abscode \sqsubseteq c.impl)$

5 Development Agreement

We define several agreements that has to be followed in order to use the reduction rule in Section 6.2. These *Agreements* are the consequences of the contract properties. they serve as the theorem to the reduction rules in Section 6.2. The component vendor (developer) guarantee that his components obey the agreement. A kind of certificate should be provided to verify its agreement.

Agreement 5.1 *Framework Invariant Implication*
$Fw.inv_I \Rightarrow Fw.inv_A$

Agreement 5.2 *Implementation refinement*
$Fw.inv_I \vdash A \sqsubseteq I$

Agreement 5.3 *Strong Invariant*

(a) $I \parallel S \parallel E \vdash sinv\ Fw.inv_I$

(b) $A \parallel S \parallel E \vdash sinv\ Fw.inv_A$

Agreement 5.4 *Environment*
A framework may forward methods from its component with additional guard.
$\mathsf{true} \vdash M \sqsubseteq E$

Agreement 5.5 *Client Script restriction*
The client script in a framework may only call methods in components within the framework. The component cannot call each other methods.
$\mathsf{true} \vdash M \sqsubseteq S$

Agreement 5.6 *Component Uniqueness*
The state space of each component is unique.

Corollary 5.7 *Disjoint*
$\mathsf{true} \vdash m_x \sqsubseteq i_y \parallel m_y$
Proof: *Follows from Agreement 5.6*
■

6 Verification

6.1 Property Semantic

The semantic of *app.props* is :

Definition 6.1 *Framework: Property semantic*
$$Fw \vdash prop_{Unity} \triangleq I \parallel S \parallel E, Fw.inv_I \vdash prop_{Unity}$$

The semantic of a property of a component contract :

Definition 6.2 *Component: Contract Property Semantic*

Safety:
$$\mathcal{C}.contract \vdash p \text{ unless } q \triangleq \mathcal{C}.abs \parallel \mathcal{C}.env, \mathcal{C}.inv_A \vdash p \text{ unless } q$$

Progress:
$$\mathcal{C}.contract \vdash p \mapsto q \triangleq \boxed{\mathcal{C}.impl} \parallel \mathcal{C}.env, \mathcal{C}.inv_I \vdash p \mapsto q$$

The symbol $\square$ means that in a parallel composition $P \parallel Q$ if the progress is relied on P, we can write $\boxed{P} \parallel Q$. In other word the action that provide the progress is taken from the P.

6.2 Reduction Rule

In the framework, There are three kinds of property: Safety, Progress by Script and Progress by Component. The reduction rule will be devided into three main rules respectively.

For safety property ($prop_{safety}$) such as *stable*, *unless*, we have:

Rule 6.3 *Framework Unless*
$$\frac{A \parallel S \parallel E, Fw.inv_A \vdash prop_{safety}}{Fw \vdash prop_{safety}}$$

Proof:
$Fw \vdash prop_{safety}$
$=$ *By Definition 6.1*
$I \parallel S \parallel E, Fw.inv_I \vdash prop_{safety}$
$\Leftarrow$ *¡Theorem A.1 and A.4*
where other assumptions follow from Agreement 5.1, 5.2 and 5.3¿
$A \parallel S \parallel E, Fw.inv_A \vdash prop_{safety}$
■

The progress property in general, cannot be decompose. Within this framework we manage to provide two kinds of progress reduction. The first is *ProgressbyScript*. It is the client script that responsible for the progress.

Rule 6.4 *Progress by Script*
$$\frac{A \parallel \boxed{S} \parallel E, Fw.inv_A \vdash p \mapsto q}{Fw \vdash p \mapsto q}$$

Proof:
$Fw \vdash p \mapsto q$
$=$ *Definition 6.1*
$I \parallel S \parallel E, Fw.inv_I \vdash p \mapsto q$
$\Leftarrow$ *Theorem A.13*
$I \parallel \boxed{S} \parallel E, Fw.inv_I \vdash p \mapsto q$
$\Leftarrow$ *Theorem A.8, Agreement 5.2, 5.1.*
$A \parallel \boxed{S} \parallel E, Fw.inv_A \vdash p \mapsto q$
■

The rule says : having a progress property in a framework, it is possible to derive the proof from the proof of its client Script S.

The second progress is progress by component,

Rule 6.5 *Progress by Component*
$$\frac{\boxed{i_x} \parallel m_x, inv_x \vdash p \mapsto q}{Fw \vdash p \mapsto q}$$

Proof:
$Fw \vdash p \mapsto q$
$=$ *by Definition 6.1 and Theorem A.6*
$\boxed{I} \parallel S \parallel E, Fw.inv_I \vdash p \mapsto q$
$\Leftarrow$ *by Theorem A.15*
$\boxed{I} \parallel M, Fw.inv_I \vdash p \mapsto q$
$\Leftarrow$ *by Theorem A.17*
$\boxed{i_x} \parallel m_x, inv_x \vdash p \mapsto q$
■

7 Discussion

This research attempted to provide an easier formalism for component verification. Based on environment restriction, the formalism preserves the verification of the components. This formalism leads us to believe in a secure component used in a framework, as long as

the component vendor provides a legitimate certificate. This formalism is an application of work on UNITY theory of composition in [9]. Compare to [7], this research is a complementary work on component testing. The certificate in the contract may also contain ranked reliability of components as mentioned by Morris. The ranked reliability could be used to value the properties verification. This may give priority on verification tasks when the proof obligation becomes too much.

The proofs in this formalism are simple. It is possible due to the environment encapsulation of the component. The environment restriction excludes any intra component interaction and eliminates the possibility of generating tightly mixed verification conditions, which are difficult to prove. We acknowledge that it requires an example to show how it works. However, due to the space limitation it is difficult to present it without further detail. Interested reader may refer to [1] for the example.

With respect to Broy formalism, which contains internal channel, our formalism is limited. We recommend more comparison research on other examples or other types of application on this framework formalism. The comparison can provide a better view of each formalism and identify the strength of each.

The word "lightweight" in the formalism name indicates that the framework is intended for a lightweight application. More comparison and investigation should provide a standard classification of lightweight formalism. We may not be able to generally provide a full correctness guarantee of an application. However, we could provide a full correctness guarantee of a lightweight formalism of component software. The lightweight application of component software may lead us to a software market where the software vendor tends to build smaller pieces of software component, but contains complete interface information and certification of its component. This research is an ongoing research which aims to contributed on a more reliable software product in the future.

References

[1] A. Azurat and I. Prasetya. A formalism for component framework. *Jurnal Ilmu Komputer dan Teknologi Informasi*, 2005.

[2] M. Broy. Multi-view modelling of software sytems. In H. D. Van and Z. Liu, editors, *Proceedings of the Workshop on Formal Aspects of Component Software (FACS)*, 2003. Also as UNU/IIST Report no. 284.

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

[4] R. G. Hamlet, D. Mason, and D. M. Woit. Theory of software reliability based on components. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 361–370. IEEE Computer Society, 2001.

[5] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, New York, 1985.

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

[7] J. Morris, G. Lee, K. Parker, G. A. Bundell, and C. P. Lam. Software component certification. *Computer*, 34(9):30–36, Sept. 2001.

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

[9] I. Prasetya, T. Vos, A. Azurat, and S. Swierstra. A unity-based framework towards component based systems. In *Proceeding of 8th International Conference on Principles of Distributed Systems (OPODIS)*, 2004. to appear in Lecture Notes of Computer Science.

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

Appendix

A Basic Definitions and Theorems

Sub Section A.1 and A.2 are based on [9].

A.1 Refinement

Theorem A.1 *Parallel refinement I*

$$\frac{inv \vdash A \sqsubseteq P}{inv \vdash A \parallel E \sqsubseteq P \parallel E}$$

Theorem A.2 *Parallel refinement II*

$$\frac{inv_1 \vdash A_1 \sqsubseteq P_1 \quad inv_2 \vdash A_2 \sqsubseteq P_2}{inv_1 \wedge inv_2 \vdash A_1 \parallel A_2 \sqsubseteq P_1 \parallel P_2}$$

Theorem A.3 *Parallel refinement III*

$$\frac{inv_1 \vdash A \sqsubseteq P_1 \quad inv_2 \vdash A \sqsubseteq P_2}{inv_1 \wedge inv_2 \vdash A \sqsubseteq P_1 \parallel P_2}$$

A.2 Unity

Theorem A.4 *Unless Refinement*

$$\frac{\begin{array}{l} inv_Q \Rightarrow inv_B \\ inv_Q \vdash B \sqsubseteq Q \\ B, inv_B \vdash p \text{ unless } q \end{array}}{Q, inv_Q \vdash p \text{ unless } q}$$

Theorem A.5 *Unless Refinement with parallel composition*

$$\frac{\begin{array}{l} A \parallel Q, i \vdash p \text{ unless } q \\ inv_P \Rightarrow inv_A \\ inv_P \vdash A \sqsubseteq P \\ P \parallel Q \vdash sinv\ inv_P \end{array}}{P \parallel Q, inv_P \vdash p \text{ unless } q}$$

Theorem A.6 *Driven Leads-to*

$$\frac{\boxed{P} \parallel Q, inv \vdash p \mapsto q}{P \parallel Q, inv \vdash p \mapsto q}$$

Theorem A.7 *Leads-to Refinement*

$$\frac{inv_P \Rightarrow inv_Q \quad inv_Q \vdash P \sqsubseteq Q \quad P, inv_P \vdash p \mapsto q}{\boxed{P} \parallel Q, inv_q \vdash p \mapsto q}$$

Theorem A.8 *Leads-to Environment Refinement*

$$\frac{\begin{array}{l}\boxed{P} \parallel B, inv_B \vdash p \mapsto q \\ inv_Q \Rightarrow inv_B \\ inv_Q \vdash B \sqsubseteq Q \\ P \parallel Q \vdash sinv\ inv_Q\end{array}}{\boxed{P} \parallel Q, inv_Q \vdash p \mapsto q}$$

Theorem A.9 *Ensures: Generalize Select*

$$\frac{\boxed{P} \parallel Q \parallel R, I \vdash p\ ensures\ q}{\boxed{P \parallel Q} \parallel R, I \vdash p\ ensures\ q}$$

where: P,Q and R, are processes with disjoint state spaces.

Theorem A.10 *Leads-to Select Refinement*

$$\frac{\begin{array}{l}\boxed{P} \parallel R, I \vdash p \mapsto q \\ J \Rightarrow I \\ P \parallel Q \parallel R \vdash sinv\ J\end{array}}{\boxed{P \parallel Q} \parallel R, J \vdash p \mapsto q}$$

where: P,Q and R, are processes with disjoint state spaces. I and J are invariants.

A.3 Component Theorems

Theorem A.11 *Contract Implication*

$$\frac{\mathcal{C}.contract \vdash p\ unless\ q}{\mathcal{C}.impl \parallel \mathcal{C}.env, inv_I \vdash p\ unless\ q}$$

Proof:

$\mathcal{C}.contract \vdash p$ unless q

$=$ *Definition 6.2*

$\mathcal{C}.abs \parallel \mathcal{C}.env, \mathcal{C}.inv_A \vdash p$ unless q

$\Rightarrow$ *Theorem A.4, A.1 and Agreement 5.1, 5.2.*

$\mathcal{C}.impl \parallel \mathcal{C}.env, \mathcal{C}.inv_I \vdash p$ unless q

■

A.4 Framework Theorems

The three main rule in Sub Section 6.2 is based on some variant of UNITY theorems. In this sub section we discuss the theorem and its proof when it is not trivial.

A.4.1 Theorems for Safety Properties

Theorem A.12 *Framework Stable Conjuct decomposition*

$$\frac{\begin{array}{l}c_1, c_2 \in Fw.contract \\ c_1 \vdash \mathsf{stable}\ p \\ c_2 \vdash \mathsf{stable}\ q\end{array}}{Fw \vdash \mathsf{stable}\ p \wedge q}$$

Proof: *By standard Unity Theorem, with Agreement 5.6.*

■

A.4.2 Theorems for Progress by Script

Theorem A.13 *Framework Progress driven by script*

$$\frac{I \parallel \boxed{S} \parallel E, Fw.inv_I \vdash p \mapsto q}{I \parallel S \parallel E, Fw.inv_I \vdash p \mapsto q}$$

Proof: *By Theorem A.6.*

■

Theorem A.14 *Leads-to Abstract Refinement*

$$\frac{\begin{array}{l}A \parallel \boxed{S} \parallel E, Fw.inv_A \vdash p \mapsto q \\ Fw.inv_I \Rightarrow Fw.inv_A \\ Fw.inv_I \vdash A \sqsubseteq I \\ I \parallel S \parallel E \vdash sinv\ Fw.inv_I\end{array}}{I \parallel \boxed{S} \parallel E, Fw.inv_I \vdash p \mapsto q}$$

Proof:

by Theorem A.8 and Theorem A.1 of $A \parallel E \sqsubseteq I \parallel E$ *and Asosiativity of* $\parallel$.

■

A.4.3 Theorems for Progress by Component

Theorem A.15 *Leads-to Method refinement*

$$\frac{\boxed{I} \parallel M, Fw.inv_I \vdash p \mapsto q}{\boxed{I} \parallel S \parallel E, Fw.inv_I \vdash p \mapsto q}$$

Proof:

$\boxed{I} \parallel S \parallel E, Fw.inv_I \vdash p \mapsto q$

$\Leftarrow$ *Theorem A.8*

$\boxed{I} \parallel M, Fw.inv_I \vdash p \mapsto q$

$Fw.inv_I \vdash M \sqsubseteq S \parallel E$

$I \parallel S \parallel E \vdash sinv\ Fw.inv_I$

$Fw.inv_I \Rightarrow Fw.inv_A$

$\Leftarrow$ *By invariant implication (Agreement 5.1), strong invariant (Agreement 5.3), and method refined (Agreement 5.4, 5.5)*

$\boxed{I} \parallel M, Fw.inv_A \vdash p \mapsto q$

■

Theorem A.16 *Pair of component selection*

$$\frac{\boxed{i_1} \parallel m_1, inv_1 \vdash p \mapsto q}{\boxed{i_1 \parallel i_2} \parallel m_1 \parallel m_2, inv_1 \wedge inv_2 \vdash p \mapsto q}$$

where $i_x = c_x.impl$
$m_x = c_x.meth$
$inv_x = c_x.impl.inv$

Proof:

$\boxed{i_1 \parallel i_2} \parallel m_1 \parallel m_2, inv_1 \wedge inv_2 \vdash p \mapsto q$
$\Leftarrow$ *Theorem A.10*
$i_1 \parallel m_1, inv_1 \vdash p \mapsto q$
$inv_1 \wedge inv_2 \vdash m_1 \sqsubseteq i_2 \parallel m_1 \parallel m_2$
$i_1 \parallel i_2 \parallel m_1 \parallel m_2 \vdash sinv\ inv_1 \wedge inv_2$
$\Leftarrow$ *Agreement 5.6 and5.3*
$\boxed{i_1} \parallel m_1, inv_1 \vdash p \mapsto q$

■

Theorem A.17 *Component Selection (generalisation)*

$$\frac{\boxed{i_x} \parallel m_x, inv_x \vdash p \mapsto q}{\boxed{I} \parallel M, Fw.inv_I \vdash p \mapsto q}$$

Proof: *By generalisation of Theorem A.16.*

■