Developing Components in the Presence of Re-entrance

Leonid Mikhajlov

Turku Centre for Computer Science,
Lemminkäisenkatu 14A, Turku
20520, Finland;
Leonid.Mikhajlov@abo.fi

Emil Sekerinski

McMaster University,
1280 Main Street West, Hamilton,
Ontario, Canada, L8S4K1;
Emil.Sekerinski@mcmaster.ca

Linas Laibinis

Turku Centre for Computer Science,
Lemminkäisenkatu 14A, Turku
20520, Finland;
Linas.Laibinis@abo.fi

Turku Centre for Computer Science
TUCS Technical Report No 239
February 1999

ISBN 952-12-0382-X
ISSN 1239-1891

Abstract

Independent development of components according to their specifications is complicated by the fact that a thread of control can exit and re-enter the same component. This kind of re-entrance may cause problems as the internal representation of a component can be observed in an inconsistent state. We argue that the ad-hoc reasoning used in establishing conformance of components to their specifications that intuitively appears to be correct does not account for the presence of re-entrance. Such reasoning leads to a conflict between assumptions that component developers make about the behaviour of components in a system, resulting in the component re-entrance problem. We formulate the modular reasoning property that captures the process of independent component development and introduce two requirements that must be imposed to avoid the re-entrance problem. Then we define a customised theory of components, component systems, and component refinement which models the process of component development from specifications. Using this theory, we prove that the formulated requirements are sufficient to establish the modular reasoning property.

Keywords: software components, modular reasoning, software correctness, specification and verification methodology, component re-entrance

TUCS Research Group
Programming Methodology Research Group

1 Introduction

In this paper we study the problem which hinders the development of a component market. One of the characteristic features of component-based systems and standards is the fact that components are developed by independent developers and an integration phase is either completely absent or minimised. When the integration phase is missing as, e.g., in CI Labs OpenDoc [7], components are composed by end users; when the integration phase is postponed, as in the case of Sun Java Beans [15] and Microsoft COM [14], components are composed by application developers. With both composition scenarios, components communicate by invoking each other's methods through the interfaces they implement. Interfaces are syntactic and only syntactic compatibility of components implementing them can be verified in the integration phase. It has been recognised [9, 16] that the verification of syntactic compatibility is insufficient to guarantee seamless interoperation of components in the resulting system. Interfaces should be augmented with behavioural specifications of the expected functionality to stipulate the contractual obligations that the components implementing such interfaces are required to meet. Due to the missing integration phase, it becomes impossible to analyse semantic integrity of the composed system. Therefore, a specification and verification method should provide for modular reasoning: verifying that participating components meet their contractual obligations should be sufficient to guarantee that the composed system operates correctly.

The independent development of components according to their specifications is complicated by the fact that, in general, a thread of control can exit and re-enter the same component. Suppose that we have two communicating components $\mathcal{A}$ and $\mathcal{B}$, each with its own attributes. A method of component $\mathcal{A}$ invokes a method of component $\mathcal{B}$. At the moment when the method of $\mathcal{B}$ is invoked, instance variables of $\mathcal{A}$ might be in transition between consistent states. The component $\mathcal{B}$ can observe and modify the state of $\mathcal{A}$ by calling back its methods. Such a re-entering method invocation pattern is problematic because $\mathcal{B}$ can observe $\mathcal{A}$ in an unexpected inconsistent state and become invalidated. Further on we refer to this problem as the *component re-entrance problem.* In order to show the implications of the component re-entrance problem on the independent development of components, we analyse the example in Fig. 1.

Let us first remark on the specification notation that we use in our examples. It was pointed out [12, 3, 9, 4, 5] that a specification can be viewed as an abstract program. In fact, specifications differ from executable programs only by the degree of nondeterminism and data structures that are used. Typically, an executable program is just a deterministic specification operating on implementable data structures. Such an approach to formal specification is advantageous, because it permits to include method calls in

$$
\begin{array}{ll}
\textbf{component } \mathit{Model} & \textbf{component } \mathit{View} \\
\quad s : \ \mathit{seq\ of\ char} := \langle\rangle & \quad \mathit{update}() \ \widehat{=} \\
\quad \mathit{get\_s}() \ \widehat{=} \ \textbf{return } s, & \qquad \mathit{print}(\#\ \mathit{Model} \to \mathit{get\_s}()) \\
\quad \mathit{get\_num}() \ \widehat{=} \ \textbf{return } \#s, & \textbf{end} \\
\quad \mathit{append}(\textbf{val } t : \ \mathit{seq\ of\ char}) \ \widehat{=} & \\
\qquad s := s \frown t; \mathit{View} \to \mathit{update}() & \\
\textbf{end} &
\end{array}
$$

Figure 1: Specification of the Model-View component system. The operator # returns the length of a sequence and the operator $\frown$ concatenates two sequences.

a component specification to fix a certain communication protocol. This approach is state-based, and in order to specify the behaviour of component methods we need to model data attributes of this component. Even though component attributes are present in its specification, they cannot be accessed by clients of this component and, therefore, can be changed in a development step. When such a change is made, component methods must be modified to work with the new attributes.

As the problem that we consider does not depend on a programming language, in the example in Fig. 1 we use a simple specification notation which should appeal to the reader's intuition. Weakest precondition and operational semantics for statements in this notation can be found in [3]. Note that each statement explicitly indicates which variables it modifies; the other variables remain unchanged.

The example follows the Observer pattern [8], which allows separating the presentational aspects of the user interface from the underlying application data, by defining two components *Model* and *View*. The components *Model* and *View* refer to each other. Note that we deliberately abstract away from the mechanism by which such mutual reference can be achieved, because we want to keep our component model as general as possible. Components can be static entities, such as modules, or dynamic entities, such as objects. In the case of static entities mutual reference can be established by mutual inclusion of syntactic interfaces, whereas with dynamic entities it can be achieved, for example, by passing pointers to components as method parameters.

The specification *Model* maintains a string s, represented by a sequence of characters and initialised with an empty sequence. Every time a new string is appended to the string in *Model*, the method *update* of *View* is called. In turn, *update* calls back *Model*'s $\mathit{get\_s}()$ method and prints out the number of elements in the received string. In a component market, these specifications are published and independent developers are offered to implement these components.

Suppose that one software company decides to implement the specification of *Model*. To avoid counting characters in the method *get_num*, they introduce an integer attribute n to represent the number of characters in the sequence. Accordingly, they implement a component *Model'* as follows:

$$
\begin{array}{l}
\textbf{component } \mathit{Model}' \\
\quad s : \ \mathit{seq\ of\ char} := \langle\rangle \\
\quad n : \mathit{int} := 0 \\
\quad \mathit{get\_s}() \ \widehat{=} \ \textbf{return } s, \\
\quad \mathit{get\_num}() \ \widehat{=} \ \textbf{return } n, \\
\quad \mathit{append}(\textbf{val } t : \ \mathit{seq\ of\ char}) \ \widehat{=} \\
\qquad s := s \frown t; \mathit{View} \rightarrow \mathit{update}(); n := n + \#t \\
\textbf{end}
\end{array}
$$

Note that taking into account the specification of the method *update* in *View*, the implementation of the method *append* appears to be perfectly valid. In fact, updating the screen as early as possible is a reasonable policy.

Now suppose that another software company decides to implement a component *View'* according to the specification. Note that the developers of *View'* do not have access to the code of *Model'*, so the only thing they can rely on is its specification. To avoid passing a sequence of characters as a parameter, the method *update* can be implemented to invoke the method *get_num* of *Model*:

$$
\begin{array}{l}
\textbf{component } \mathit{View}' \\
\quad \mathit{update}() \ \widehat{=} \ \mathit{print}(\mathit{Model} \rightarrow \mathit{get\_num}()) \\
\textbf{end}
\end{array}
$$

Here we face the component re-entrance problem. Even though individual components appear to implement the specification correctly, their composition is apparently incorrect with respect to the same specification: the number of elements in the string s that *update* prints out is wrong.

In a component market, where developments have to be independent, this constitutes a major obstacle. However, if we view *Model* and *View* as implementations and *Model'* and *View'* as more efficient implementations, we see that this problem occurs not only during development, but also during maintenance of component systems. The formalism in which we study the problem encompasses both situations in a uniform way.

A recommendation known from practice suggests always to establish a component invariant before the thread of control leaves the component. In fact, this is the recommendation for implementing the Observer pattern as it can be found in [8]. In the example above, the developers of *Model'* should have established the component invariant $n = \#s$ before invoking the method *update*.

In this paper we present a formal analysis of the problem that supports this requirement, but reveals that it is not sufficient in the general case. Two further restrictions should be imposed according to the "no call-back assumptions" and the "no accidental mutual recursion" requirements.

The rest of the paper is organised as follows. We begin with a detailed analysis of the component re-entrance problem and explain why we view it as the conflict of assumptions that developers of components make about the behaviour of other components in the system. We formulate the modular reasoning property that captures the process of independent component development. Using simple examples, we then justify the introduction of two requirements that must be imposed to avoid the re-entrance problem. Next we develop a customised theory of components, component composition, and refinement and prove a modular reasoning theorem which states that the modular reasoning property reinforced with our requirements holds in the presence of re-entrance. Finally, we offer a discussion of implications of the modular reasoning theorem, discuss related work, and provide some insights on our future work.

2 The Essence of The Component Re-entrance Problem

A component operates by communicating with an environment. Unlike in the case of procedure libraries, the environment calls back the component's methods. The component and its environment play symmetrical roles: the component is a client of the environment, while the environment is a client of the component. Therefore, we can view the entire system as consisting of only two components, the component under consideration and the component "environment".

Let us now define the notion of behavioural conformance more precisely. We say that a system (or a component) $\mathbf{S}$ *is refined by* a system (or a component) $\mathbf{S}'$ if the externally observable behaviour of $\mathbf{S}'$ is the externally observable behaviour of $\mathbf{S}$ or an improvement of it. In other words, if $\mathbf{S}'$ is a refinement of $\mathbf{S}$ then it is substitutable for $\mathbf{S}$ in any context.[1] Note that $\mathbf{S}$ and $\mathbf{S}'$ can be, respectively, a specification and a more concrete specification, a specification and an implementation, an implementation and a more efficient implementation.

Now suppose that we have a specification of a system composed of two components $\mathcal{A}$ and $\mathcal{B}$ invoking each other's methods. Ultimately, independent developers of refining components $\mathcal{A}'$ and $\mathcal{B}'$ would like to achieve that the system resulting from the composition of these components be a refine-

[1] The formal definition of refinement is given in Sec.4.3.

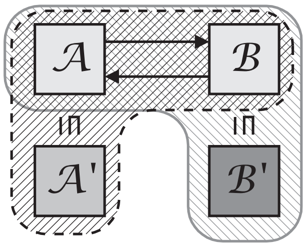

Figure 2: Independence of component development. Developers can access only the components in the corresponding hatched areas.

ment of the composition of the original components $\mathcal{A}$ and $\mathcal{B}$, namely,

$$\mathcal{A}\,\mathbf{comp}\,\mathcal{B} \textit{ is refined by } \mathcal{A}'\,\mathbf{comp}\,\mathcal{B}' \tag{1}$$

where **comp** composes two components into a component system. A composition of two components has all the methods of both components with all mutual method calls resolved. Due to the late integration phase, which is characteristic of component systems, developers of a component cannot analyse the source code of the new environment this component will be used in, and can rely only on original specification of the system. This setting is illustrated in Fig. 2.

The behaviour of a component invoking methods of another component depends on the behaviour of these methods. Therefore, when reasoning about the conformance of the component $\mathcal{A}'$ to the component $\mathcal{A}$, the developers need to make assumptions about the behaviour of the component $\mathcal{B}$. The ad-hoc method for taking such assumptions into account is to reason about the refinement between the results of composition of $\mathcal{A}'$ and $\mathcal{A}$ with $\mathcal{B}$:

$$\mathcal{A}\,\mathbf{comp}\,\mathcal{B} \textit{ is refined by } \mathcal{A}'\,\mathbf{comp}\,\mathcal{B}, \tag{2}$$

and dually for the components $\mathcal{B}'$ and $\mathcal{B}$:

$$\mathcal{A}\,\mathbf{comp}\,\mathcal{B} \textit{ is refined by } \mathcal{A}\,\mathbf{comp}\,\mathcal{B}' \tag{3}$$

Unfortunately, in the general case, these two requirements (2) and (3) are insufficient to establish goal (1), as demonstrated by the previous example. In other words, the desired property

$$\begin{array}{ll} \textit{if} & \mathcal{A}\,\mathbf{comp}\,\mathcal{B} \textit{ is refined by } \mathcal{A}'\,\mathbf{comp}\,\mathcal{B} \textit{ and} \\ & \mathcal{A}\,\mathbf{comp}\,\mathcal{B} \textit{ is refined by } \mathcal{A}\,\mathbf{comp}\,\mathcal{B}' \\ \textit{then} & \mathcal{A}\,\mathbf{comp}\,\mathcal{B} \textit{ is refined by } \mathcal{A}'\,\mathbf{comp}\,\mathcal{B}' \end{array} \tag{4}$$

does not hold. We believe that this fact constitutes the essence of the component re-entrance problem. The problem occurs due to the conflict of assumptions the developers of components make about the behaviour of other components in the system. In the previous example the developers of the component *View'* assumed that at the moment when the method *update* is called the invariant of the implementation of *Model* would hold. Similarly, the developers of *Model'* assumed that they do not need to establish the invariant before invoking *update*, because its specification does not rely on it. These conflicting assumptions led to the problem during composition.

This consideration brings us to the questions, how we can guide the process of component development, so that the system composed of the refining components would always be safely substitutable for the original one, and how while developing a component one can make assumptions about the behaviour of the other components in the system, in a consistent manner.

3 Modular Reasoning Required

Apparently it would be desirable if for establishing refinement between composed systems it would be sufficient to verify refinement between the corresponding components. In other words, we would like the following property to hold:

$$\begin{array}{l} \quad\textit{if} \;\; \mathcal{A} \textit{ is refined by } \mathcal{A}' \textit{ and} \\ \quad\quad\;\; \mathcal{B} \textit{ is refined by } \mathcal{B}' \\ \textit{then } \mathcal{A} \,\mathbf{comp}\, \mathcal{B} \textit{ is refined by } \mathcal{A}' \,\mathbf{comp}\, \mathcal{B}' \end{array}$$

However, establishing refinement between the participating components is complicated due to their mutual dependence. In principle, we can say that a component is refined by another component if the systems resulting from the composition of these components with an arbitrarily chosen component are in refinement:

$$\mathcal{A} \textit{ is refined by } \mathcal{A}' \;\widehat{=}\; \mathcal{A} \,\mathbf{comp}\, \mathcal{B} \textit{ is refined by } \mathcal{A}' \,\mathbf{comp}\, \mathcal{B} \textit{ for any } \mathcal{B}$$

In fact, it is possible to prove that this definition of refinement indeed establishes property (4) for the case of mutually dependent components. Unfortunately, this definition of refinement is too restrictive to be used in practice. According to this definition, one can only refine bodies of methods around method invocations, without being able to assume anything about the called methods.

For the definition of component refinement to be useful in practice it should permit to make assumptions about the context in which the component under consideration operates. As the context (environment) of a component can be seen as the other component, we would like the following *modular reasoning property* to hold:

$$\begin{array}{l} \quad\textit{if} \;\; \mathcal{A} \textit{ is refined by } \mathcal{A}' \textit{ in context of } \mathcal{B} \textit{ and} \\ \quad\quad\;\; \mathcal{B} \textit{ is refined by } \mathcal{B}' \textit{ in context of } \mathcal{A} \\ \textit{then } \mathcal{A} \,\mathbf{comp}\, \mathcal{B} \textit{ is refined by } \mathcal{A}' \,\mathbf{comp}\, \mathcal{B}' \end{array}$$

In the case when the complete context is assumed in a refinement step, the modular reasoning property is equivalent to property (4). However, as was demonstrated by the previous example, the conclusion of the modular reasoning property does not hold in this case. In order to establish refinement between the composed systems, it is necessary to restrict the assumptions that component developers can make about the context in which the component is going to operate in. To identify the restrictions that should be imposed on the assumptions about the component context, let us consider two counter examples invalidating property (4).

In the following example we use an assertion statement $\{p\}$, where p is a state predicate. If p is true in the current state, the assertion skips, otherwise it aborts. Therefore, the assertion statement can be seen as an abbreviation for the corresponding conditional.

component A
$\quad m_1(\textbf{valres}\ x : \mathit{int}) \ \widehat{=}\ \{x > 5\}; x := 5,$
$\quad m_2(\textbf{valres}\ x : \mathit{int}) \ \widehat{=}\ \{x > 0\}; x := 5$
end
component A'
$\quad m_1(\textbf{valres}\ x : \mathit{int}) \ \widehat{=}\ \{x > 0\}; x := 5,$
$\quad m_2(\textbf{valres}\ x : \mathit{int}) \ \widehat{=}\ B \rightarrow n(x)$
end

component B
$\quad n(\textbf{valres}\ x : \mathit{int}) \ \widehat{=}$
$\qquad A \rightarrow m_1(x)$
end
component B'
$\quad n(\textbf{valres}\ x : \mathit{int}) \ \widehat{=}$
$\qquad \{x > 5\}; x := 5$
end

If we expand the bodies of the method m_2 in the composed systems, then we have

$$(\mathcal{A}\,\textbf{comp}\,\mathcal{B}) :: m_2 = \{x > 0\}; x := 5 \quad (\mathcal{A}'\,\textbf{comp}\,\mathcal{B}) :: m_2 = \{x > 0\}; x := 5$$
$$(\mathcal{A}\,\textbf{comp}\,\mathcal{B}') :: m_2 = \{x > 0\}; x := 5 \quad (\mathcal{A}'\,\textbf{comp}\,\mathcal{B}') :: m_2 = \{x > 5\}; x := 5$$

Therefore,

$(\mathcal{A}\,\textbf{comp}\,\mathcal{B}) :: m_2$ *is refined by* $(\mathcal{A}'\,\textbf{comp}\,\mathcal{B}) :: m_2$ and
$(\mathcal{A}\,\textbf{comp}\,\mathcal{B}) :: m_2$ *is refined by* $(\mathcal{A}\,\textbf{comp}\,\mathcal{B}') :: m_2$

However, it is not the case that

$(\mathcal{A}\,\textbf{comp}\,\mathcal{B}) :: m_2$ *is refined by* $(\mathcal{A}'\,\textbf{comp}\,\mathcal{B}') :: m_2$

Due to the presence of assertions, the precondition $x > 5$ of $(\mathcal{A}'\,\textbf{comp}\,\mathcal{B}') :: m_2$ is stronger than the precondition $x > 0$ of $(\mathcal{A}\,\textbf{comp}\,\mathcal{B}) :: m_2$, while to preserve refinement, preconditions can only be weakened.

This example motivates us to formulate the following "no call-back assumptions" requirement:

> *While developing an implementation of a method, implementations of other methods of the same component cannot be assumed; their specifications should be considered instead.*

As the behaviour of the environment serving as a context depends on the behaviour of the component under consideration, assuming that the environment is going to call back on the refined component would implicitly modify the specification.

However, there exists another aspect of the component re-entrance problem which cannot be handled by simply restricting the context for refinement. The following rather trivial example illustrates this aspect of the problem.

component A	**component** B
$\quad m(\mathbf{res}\ r : int) \ \widehat{=}\ r := 5$	$\quad n(\mathbf{res}\ r : int) \ \widehat{=}\ r := 5$
end	**end**
component A'	**component** B'
$\quad m(\mathbf{res}\ r : int) \ \widehat{=}\ B \rightarrow n(r)$	$\quad n(\mathbf{res}\ r : int) \ \widehat{=}\ A \rightarrow m(r)$
end	**end**

It is easy to see that a call to any method in the composition $\mathcal{A}'\,\mathbf{comp}\,\mathcal{B}'$ of the refined components leads to a never terminating recursion of method invocations. Obviously, such a behaviour does refine the behaviour of the original system. Based on this example, we formulate the following "no accidental mutual recursion" requirement:

> *Independent development of components should not introduce unexpected mutual recursion.*

We claim that if the "no call-back assumptions" and "no accidental mutual recursion" requirements are satisfied, then the modular reasoning property holds. For proving this claim formally we develop a customised theory of components, component systems, and their refinement.

4 Formalisation of Components, Composition, and Refinement

We formalise components, component systems, and refinement between them within the refinement calculus. For simplicity we assume that components do not have self-calls and component implementations do not introduce new methods. Our model is tailored specifically to allow for reasoning about the properties under consideration.

4.1 Statements and Statement Refinement

This subsection is based on the work by Ralph Back and Joakim von Wright as presented in [2, 3]. The refinement calculus is a logical framework for reasoning about correctness and refinement of imperative programs. The language used to express programs and specifications is essentially Dijkstra's language of guarded commands, with some extensions. Each command of this language is identified with its weakest precondition predicate transformer. Therefore, program statements are modelled as functions that map postconditions to preconditions.

The predicates over a state space (type) Σ are functions from Σ to *Bool*, denoted by $\mathcal{P}\Sigma$. The relations from Σ to Γ are functions from Σ to a predicate (set of values) over Γ, denoted by $\Sigma \leftrightarrow \Gamma$. The predicate transformers from Σ to Γ are functions mapping predicates over Γ to predicates over Σ, denoted by $\Sigma \mapsto \Gamma$ (note the reversion of the direction), or by $Ptran(\Sigma)$ in the case of $\Sigma \mapsto \Sigma$.

The entailment ordering $p \subseteq q$ on predicates $p, q : \mathcal{P}\Sigma$ is defined as universal implication on booleans, i.e.

$$p \subseteq q \;\;\widehat{=}\;\; (\forall \sigma : \Sigma \bullet p.\sigma \;\Rightarrow\; q.\sigma)$$

The conjunction and disjunction on predicates $\cup$ and $\cap$ are defined pointwise. The predicates *true* and *false* over Σ map every $\sigma : \Sigma$ to the boolean values $\mathbf{T}$ and $\mathbf{F}$, respectively. The refinement ordering $S \sqsubseteq T$, read S *is refined by* T, on statements $S, T : \Sigma \mapsto \Gamma$ is defined by universal entailment:

$$S \sqsubseteq T \;\;\widehat{=}\;\; (\forall q : \mathcal{P}\Gamma \bullet S.q \;\subseteq\; T.q)$$

A predicate transformer $S : \Sigma \mapsto \Gamma$ is said to be *monotonic* if for all predicates p and q, $p \subseteq q$ implies $S.p \subseteq S.q$. Statements from Σ to Γ are identified with monotonic predicate transformers from Σ to Γ. Statements of this kind may be concrete, i.e. executable, or abstract, i.e. specifications. The refinement calculus includes all standard program statements, such as assignments, conditionals, and loops. Here we only present the definitions of the constructs that are used later.

The sequential composition of statements $S : \Sigma \mapsto \Gamma$ and $T : \Gamma \mapsto \Delta$ is modelled by their functional composition, for $q : \mathcal{P}(\Delta)$,

$$(S;T).q \;\;\widehat{=}\;\; S.(T.q)$$

The statement **abort** does not guarantee any outcome or termination, therefore, it maps every postcondition to *false*. The statement **magic** is *miraculous*, since it is always guaranteed to establish any postcondition. The statement **skip** leaves the state unchanged. Thus, we have:

$$\mathbf{abort}.q \;\widehat{=}\; \mathit{false} \qquad\qquad \mathbf{magic}.q \;\widehat{=}\; \mathit{true} \qquad\qquad \mathbf{skip}.q \;\widehat{=}\; q$$

The *assertion* statement $\{p\}$ indicates that predicate p is known to hold at a certain point in the program. The assertion $\{p\}$ behaves as **abort** if p does not hold, and as **skip** otherwise. Formally, it is defined as follows:

$$\{p\}.q \;\widehat{=}\; p \;\cap\; q$$

The language supports two kinds of non-deterministic updates which, in fact, represent specification statements. Given a relation $P : \Sigma \leftrightarrow \Gamma$, the *angelic update* $\{P\} : \Sigma \mapsto \Gamma$, and the *demonic update* $[P] : \Sigma \mapsto \Gamma$ are defined by

$$\begin{array}{lcl} \{P\}.\, q.\, \sigma & \hat{=} & (\exists\, \gamma : \Gamma \bullet P.\, \sigma.\, \gamma \ \wedge\ q.\, \gamma) \\ [P].\, q.\, \sigma & \hat{=} & (\forall \gamma : \Gamma \bullet P.\, \sigma.\, \gamma \ \Rightarrow\ q.\, \gamma) \end{array}$$

When started in a state σ, $\{P\}$ angelically chooses a new state γ such that $P.\, \sigma.\, \gamma$ holds, while $[P]$ demonically chooses a new state γ such that $P.\, \sigma.\, \gamma$ holds. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ behaves as **magic**. Traditional pre-postcondition specifications can be easily expressed in the refinement calculus. For example, a specification with the precondition $x > 0$ and postcondition $x' > x$, where x' stands for the new value of the program variable x, can be expressed by the statement $\{p\}; [P]$, where $p.\, x\ =\ x > 0$ and $P.\, x.\, x'\ =\ x' > x$.

The cartesian product of state spaces Σ and Γ is written $\Sigma \times \Gamma$. For predicates $p : \mathcal{P}\Sigma$ and $q : \mathcal{P}\Gamma$, their product $p \times q$ is a predicate of type $\mathcal{P}(\Sigma \times \Gamma)$ defined by

$$(p \times q).\, (\sigma, \gamma) \ \ \hat{=}\ \ p.\, \sigma\ \wedge\ q.\, \gamma$$

For relations $P_1 : \Sigma_1 \leftrightarrow \Gamma_1$ and $P_2 : \Sigma_2 \leftrightarrow \Gamma_2$, their product $P_1 \times P_2$, is a relation of type $(\Sigma_1 \times \Sigma_2) \leftrightarrow (\Gamma_1 \times \Gamma_2)$, where for $\sigma_1 : \Sigma_1$, $\sigma_2 : \Sigma_2$, $\gamma_1 : \Gamma_1$, and $\gamma_2 : \Gamma_2$, we have:

$$(P_1 \times P_2).\, (\sigma_1, \sigma_2).\, (\gamma_1, \gamma_2) \ \ \hat{=}\ \ (P_1.\, \sigma_1.\, \gamma_1)\ \wedge\ (P_2.\, \sigma_2.\, \gamma_2)$$

For predicate transformers $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$, their product $S_1 \times S_2$ is a predicate transformer of type $\Sigma_1 \times \Sigma_2 \ \mapsto\ \Gamma_1 \times \Gamma_2$ whose execution has the same effect as the simultaneous execution of S_1 and S_2:

$$(S_1 \times S_2).\, q \ \ \hat{=}\ \ (\cup\, q_1, q_2 \,|\, q_1 \times q_2 \subseteq q \bullet S_1.\, q_1 \times S_2.\, q_2)$$

The cross products operators are not associative in the sense that e.g. $S_1 \times (S_2 \times S_3) \neq (S_1 \times S_2) \times S_3$. As different associations of the cross product operators are isomorphic to each other, for simplicity we disregard the non-associativity.

Statements S and S' operating on state spaces Σ and Σ', respectively, can be combined using a relation $R : \Sigma' \leftrightarrow \Sigma$ which, when lifted to predicate transformers, gives the update statements $\{R\} : \Sigma' \mapsto \Sigma$ and $[R^{-1}] : \Sigma \mapsto \Sigma'$. Statement $S{\downarrow}R$ operates on the state space Σ', where the *concrete coercion* ${\downarrow}R$ is a convenient abbreviation given as follows:

$$S{\downarrow}R \ \ \hat{=}\ \ \{R\}; S; [R^{-1}]$$

For lists of statements, concrete and abstract coercions are defined pointwise.

Coercions can be used to define *data refinement* in terms of ordinary refinement [18]. A statement $S : Ptran(\Sigma)$ is data refined by a statement $S' : Ptran(\Sigma')$ via a relation $R : \Sigma' \leftrightarrow \Sigma$, connecting concrete and abstract states, if S, concretely coerced with R is refined by S', i.e.

$$S \sqsubseteq_R S' \;\widehat{=}\; S{\downarrow}R \sqsubseteq S'$$

A statement is said to be *indifferent* with respect to a coercion relation if it does not operate on the state component coerced with the relation. An indifferent statement **skip** $\times\, S$ is characterised by the following property:

$$(\mathbf{skip} \times S){\downarrow}(R \times Id) \sqsubseteq (\mathbf{skip} \times S)$$

Relations of the form $R \times Id$ and $Id \times P$ are said to be *orthogonal* to each other. Further on, we use the following property of concrete coercion for the orthogonal relations $R \times Id$ and $Id \times P$:

$$S{\downarrow}(R \times Id){\downarrow}(Id \times P) = S{\downarrow}(Id \times P){\downarrow}(R \times Id) = S{\downarrow}(P \times R)$$

Forward functional composition is denoted by $\circ$ and defined in the usual way:

$$(f \circ g).\, x \quad \widehat{=} \quad f.\,(g.\, x)$$

Repeated function application f^n is defined inductively by

$$\begin{array}{lcl} f^0.\, x & = & x \\ f^{n+1}.\, x & = & f^n.\,(f.\, x) \end{array}$$

4.2 Components and Composition

As we have mentioned, any component system can be seen as consisting of two components $\mathcal{A}$ and $\mathcal{B}$. Suppose that $\mathcal{A}$ has m and $\mathcal{B}$ has n methods. The components communicate by invoking each other's methods and passing parameters. For simplicity, we model method parameters by global variables that methods of both components can access in turns. For every formal parameter of a method we introduce a separate global variable which is used for passing values in and out of components. It is easy to see that parameter passing by value and by reference can be modelled in this way. As due to encapsulation the type of the internal state of the other component is not known, we say that the body of a method of the component $\mathcal{A}$ has the type $Ptran(\Sigma \times \Delta \times \beta)$, where Σ is the type of $\mathcal{A}$'s internal state, Δ is the type of global variables modelling method parameters, and β is the type variable to be instantiated with the type of the internal state of the other component during composition. As the internal state of the other component is not accessible, we assume that methods of $\mathcal{A}$ operate only on their internal state and the state representing method parameters and are, therefore, of the form $S \times \mathbf{skip}$. Similarly, methods of $\mathcal{B}$ have bodies that are of the form $\mathbf{skip} \times S$ and of the type $Ptran(\alpha \times \Delta \times \Gamma)$, where α is the type variable.

The behaviour of a component method depends on the behaviour of the methods it invokes. We can model a method of the component $\mathcal{A}$ as a function of a list of method bodies returning a method body [2]:

$$a_i \;\widehat{=}\; \lambda Bb \bullet ab_i$$

If we introduce an abbreviation $\mathcal{L}_n(\Psi)$ to stand for $\Psi \times ... \times \Psi$ with n occurrences of Ψ, we can write out the type of a_i as $\mathcal{L}_n(Ptran(\Sigma \times \Delta \times \beta)) \rightarrow Ptran(\Sigma \times \Delta \times \beta)$, where n is the number of methods of $\mathcal{B}$. Methods of $\mathcal{B}$ are defined in the same manner, but have the type $\mathcal{L}_m(Ptran(\alpha \times \Delta \times \Gamma)) \rightarrow Ptran(\alpha \times \Delta \times \Gamma)$, where m is the number of methods in $\mathcal{B}$. We assume that every method is monotonic in its argument. Accordingly, we can collectively describe all methods of $\mathcal{A}$ as a function A given as follows:

$$\begin{aligned} A \;\widehat{=}\; & (\lambda Bb \bullet (ab_1, ..., ab_m)) : \\ & \quad \mathcal{L}_n(Ptran(\Sigma \times \Delta \times \beta)) \rightarrow \mathcal{L}_m(Ptran(\Sigma \times \Delta \times \beta)) \end{aligned}$$

Therefore, the component $\mathcal{A}$ is a tuple (a_0, A), where $a_0 : \Sigma$ is an initial value of the internal state and A is the function as defined above. The definition of the component $\mathcal{B}$ is similar but with the corresponding differences in typing.

Composing components $\mathcal{A}$ and $\mathcal{B}$ results in a component system that has methods of both components with all mutual calls resolved. The methods of the component $\mathcal{A}$ in the composed system can be approximated by $A.\,B.\,\mathbf{Abort}$, where $\mathbf{Abort}$ is a tuple of **abort** statements. Using functional composition, this can be rewritten as $(A \circ B).\,\mathbf{Abort}$. Methods in such an approximation behave as the methods of $\mathcal{A}$ with all external calls redirected to $\mathcal{B}$, but with external calls of $\mathcal{B}$ aborting rather then going back to $\mathcal{A}$. Hence a better approximation of the methods of $\mathcal{A}$ in the composed system would be $(A \circ B \circ A \circ B).\,\mathbf{Abort}$, and yet a better one $(A \circ B \circ A \circ B \circ A \circ B).\,\mathbf{Abort}$, etc. The desired result is then the limit of this sequence. This limit can be expressed as the least fixed point $\mu(A \circ B)$, which is the least Xb with respect to the refinement ordering on tuples of statements such that $Xb = (A \circ B).\,Xb$. Choosing the least fixed point means that a non-terminating sequence of calls from $\mathcal{A}$ to $\mathcal{B}$ and back is equivalent to **abort**, which is the meaning of a non-terminating loop. According to the theorem of Knaster-Tarski [17], a monotonic function has a unique least fixed point in a complete lattice. Statements form a complete lattice with the refinement ordering $\sqsubseteq$ and the function $A \circ B$ is monotonic in

[2] We accept the following scheme for naming variables: a variable starting with a capital letter represents a tuple of variables; the second letter b in the name of a variable means that it represents a method body (statement) or a tuple of method bodies.

its argument, therefore, $\mu(A \circ B)$ exists and is unique. Similarly, the methods of the component $\mathcal{B}$ in the composed system are defined by $\mu(B \circ A)$.

The component system resulting from the composition of the components $\mathcal{A}$ and $\mathcal{B}$ can now be defined as follows:

$$(\mathcal{A}\,\mathbf{comp}\,\mathcal{B}) \;\widehat{=}\; ((a_0, b_0), (\mu\,(A \circ B),\; \mu\,(B \circ A)))$$

Note that during composition, the type variables α and β, representing unknown state spaces of the components $\mathcal{B}$ and $\mathcal{A}$, get instantiated with Σ and Γ respectively, so that the composed system has methods operating on the state space $\Sigma \times \Delta \times \Gamma$.

4.3 Refining Components and Component Systems

Let $A : \mathcal{L}_n(Ptran(\Sigma \times \Delta \times \beta)) \to \mathcal{L}_m(Ptran(\Sigma \times \Delta \times \beta))$ and $A' : \mathcal{L}_n(Ptran(\Sigma' \times \Delta \times \beta)) \to \mathcal{L}_m(Ptran(\Sigma' \times \Delta \times \beta))$ be methods of components $\mathcal{A}$ and $\mathcal{A}'$, respectively. We say that A is data refined by A' in the context of component $\mathcal{B}$ via a relation $R \times Id \times Id$ if

$$A \overset{\mathcal{B}}{\sqsubseteq}_R A' \;\widehat{=}\; (\mu(A \circ B))\downarrow(R \times Id \times Id) \sqsubseteq A'.\,(\mu(B \circ A)\downarrow(R \times Id \times Id))$$

For methods of components $\mathcal{B}$ and $\mathcal{B}'$ we have a similar definition but via a relation $Id \times Id \times P$.

As method bodies of the components $\mathcal{A}'$ and $\mathcal{B}'$ are indifferent to the relations $Id \times Id \times P$ and $R \times Id \times Id$ respectively, we use the following *coercion propagation lemma* :

$$\begin{array}{lcl}(A'.\,Xb)\downarrow(Id \times Id \times P) & \sqsubseteq & A'.\,(Xb\downarrow(Id \times Id \times P)) \\ (B'.\,Yb)\downarrow(R \times Id \times Id) & \sqsubseteq & B'.\,(Yb\downarrow(R \times Id \times Id))\end{array}$$

The proof of this lemma is presented in appendix.

We say that $\mathcal{A} = (a_0 : \Sigma, A : \mathcal{L}_n(Ptran(\Sigma \times \Delta \times \beta)) \to \mathcal{L}_m(Ptran(\Sigma \times \Delta \times \beta)))$ is refined by $\mathcal{A}' = (a_0' : \Sigma', A' : \mathcal{L}_n(Ptran(\Sigma' \times \Delta \times \beta)) \to \mathcal{L}_m(Ptran(\Sigma' \times \Delta \times \beta)))$ in the context of $\mathcal{B}$, if there exists a relation $R : \Sigma' \leftrightarrow \Sigma$ such that this relation holds between the initial values, and methods of $\mathcal{A}$ are data refined by methods of $\mathcal{A}'$ in the context of $\mathcal{B}$ via the relation $R \times Id \times Id$. Formally,

$$\mathcal{A} \overset{\mathcal{B}}{\sqsubseteq} \mathcal{A}' \;\widehat{=}\; (\exists R \bullet (R.\,a_0'.\,a_0) \;\wedge\; A \overset{\mathcal{B}}{\sqsubseteq}_R A')$$

For the components $\mathcal{B} = (b_0 : \Gamma, B)$ and $\mathcal{B}' = (b_0' : \Gamma', B')$ the definition of refinement is similar, only that the initial values are connected via a relation $P : \Gamma' \leftrightarrow \Gamma$ and the lists of method bodies are connected via $Id \times Id \times P$.

We say that the component system $\mathcal{A}\,\mathbf{comp}\,\mathcal{B}$ is refined by the component system $\mathcal{A}'\,\mathbf{comp}\,\mathcal{B}'$, if there exist such relations R and P that initial values of these component systems are related via the relation $R \times P$ and lists of method bodies are related via the relation $R \times Id \times P$. Formally, we have:

$$
\begin{aligned}
&\mathcal{A}\,\mathbf{comp}\,\mathcal{B} \sqsubseteq \mathcal{A}'\,\mathbf{comp}\,\mathcal{B}' \;\widehat{=} \\
&\quad (\exists R, P \bullet\; (R \times P).\,(a_0', b_0').\,(a_0, b_0) \;\wedge \\
&\qquad \mu(A \circ B){\downarrow}(R \times Id \times P) \sqsubseteq \mu(A' \circ B') \;\wedge \\
&\qquad \mu(B \circ A){\downarrow}(R \times Id \times P) \sqsubseteq \mu(B' \circ A'))
\end{aligned}
$$

5 Modular Reasoning Theorem

Our objective is to prove that the modular reasoning property holds for mutually dependent components if the "no call-back assumptions" and "no accidental mutual recursion" requirements are satisfied. First we formulate and prove the modular reasoning theorem which captures the mathematical meaning of the modular reasoning property reinforced with the requirements. Then we explain how the requirements are reflected in the assumptions of the theorem. As the "no accidental mutual recursion" requirement is non-modular, in the sense that it requires checking for the absence of mutual recursion in the system composed from refining components, we then discuss techniques which permit to satisfy this requirement in a modular fashion.

5.1 Formulating and Proving the Theorem

Modular Reasoning Theorem. *Let components* $\mathcal{A}$, $\mathcal{B}$, $\mathcal{A}'$, and $\mathcal{B}'$ *be given as follows*:

$$
\begin{aligned}
\mathcal{A} &= (a_0 : \Sigma, A : \mathcal{L}_n(Ptran(\Sigma \times \Delta \times \beta)) \to \mathcal{L}_m(Ptran(\Sigma \times \Delta \times \beta))),\\
\mathcal{B} &= (b_0 : \Gamma, B : \mathcal{L}_m(Ptran(\alpha \times \Delta \times \Gamma)) \to \mathcal{L}_n(Ptran(\alpha \times \Delta \times \Gamma))),\\
\mathcal{A}' &= (a_0' : \Sigma', A' : \mathcal{L}_n(Ptran(\Sigma' \times \Delta \times \beta)) \to \mathcal{L}_m(Ptran(\Sigma' \times \Delta \times \beta))),\\
\mathcal{B}' &= (b_0' : \Gamma', B' : \mathcal{L}_m(Ptran(\alpha \times \Delta \times \Gamma')) \to \mathcal{L}_n(Ptran(\alpha \times \Delta \times \Gamma')))
\end{aligned}
$$

Then we have:

$$
\begin{aligned}
&\mathcal{A} \overset{\mathcal{B}}{\sqsubseteq} \mathcal{A}' \;\wedge && (a)\\
&\mathcal{B} \overset{\mathcal{A}}{\sqsubseteq} \mathcal{B}' \;\wedge && (b)\\
&(\exists n \bullet\; \forall Xb \bullet\; \mu(A' \circ B') \;=\; (A' \circ B')^n.\,Xb) \;\wedge && (c)\\
&(\exists m \bullet\; \forall Yb \bullet\; \mu(B' \circ A') \;=\; (B' \circ A')^m.\,Yb) \;\Rightarrow && (d)\\
&\qquad \mathcal{A}\,\mathbf{comp}\,\mathcal{B} \sqsubseteq \mathcal{A}'\,\mathbf{comp}\,\mathcal{B}'
\end{aligned}
$$

Proof Expanding the definitions and making simple logical transformations, we get three subgoals:

$$
\begin{array}{ll}
1. & (R.\,a_0'.\,a_0)\ \wedge\ (P.\,b_0'.\,b_0)\ \Rightarrow\ (R\times P).\,(a_0',b_0').\,(a_0,b_0) \\
2. & A\overset{\mathcal{B}}{\sqsubseteq}_R A'\ \wedge\ B\overset{\mathcal{A}}{\sqsubseteq}_P B'\ \wedge\ (c)\ \wedge\ (d) \Rightarrow \\
 & \qquad \mu(A\circ B)\downarrow(R\times Id\times P)\ \sqsubseteq\ \mu(A'\circ B') \\
3. & A\overset{\mathcal{B}}{\sqsubseteq}_R A'\ \wedge\ B\overset{\mathcal{A}}{\sqsubseteq}_P B'\ \wedge\ (c)\ \wedge\ (d) \Rightarrow \\
 & \qquad \mu(B\circ A)\downarrow(R\times Id\times P)\ \sqsubseteq\ \mu(B'\circ A')
\end{array}
$$

where R and P are fixed but arbitrary relations. The first subgoal is obviously true. To prove the second and the third subgoals, we first prove the following lemma.

Lemma. *For components $\mathcal{A}, \mathcal{B}, \mathcal{A}', \mathcal{B}'$ defined as above, relations $R : \Sigma' \leftrightarrow \Sigma$* and *$P : \Gamma' \leftrightarrow \Gamma$, and any natural number n, we have*:

$$\mu(A\circ B)\downarrow(R\times Id\times P)\ \sqsubseteq\ (A'\circ B')^n.\,(\mu(A\circ B)\downarrow(R\times Id\times P))$$

Proof We prove this lemma by induction over n.
Base case:

$$
\begin{array}{ll}
 & (A'\circ B')^0.\,(\mu(A\circ B)\downarrow(R\times Id\times P)) \\
= & \{\textit{definition of } f^0\} \\
 & \mu(A\circ B)\downarrow(R\times Id\times P)
\end{array}
$$

Inductive case:
Assuming $\mu(A\circ B)\downarrow(R\times Id\times P)\ \sqsubseteq\ (A'\circ B')^n.\,(\mu(A\circ B)\downarrow(R\times Id\times P))$, we calculate:

$$
\begin{array}{ll}
 & \mu(A\circ B)\downarrow(R\times Id\times P) \\
\sqsubseteq & \{\textit{induction assumption}\} \\
 & (A'\circ B')^n.\,(\mu(A\circ B)\downarrow(R\times Id\times P)) \\
= & \{\textit{relations } R\times Id\times Id \textit{ and } Id\times Id\times P \textit{ are orthogonal}\} \\
 & (A'\circ B')^n.\,(\mu(A\circ B)\downarrow(R\times Id\times Id)\downarrow(Id\times Id\times P)) \\
\sqsubseteq & \{\textit{definition of } A\overset{\mathcal{B}}{\sqsubseteq}_R A'\} \\
 & (A'\circ B')^n.\,(A'.\,(\mu(B\circ A)\downarrow(R\times Id\times Id))\downarrow(Id\times Id\times P)) \\
\sqsubseteq & \{\textit{coercion propagation lemma}\} \\
 & (A'\circ B')^n.\,(A'.\,(\mu(B\circ A)\downarrow(R\times Id\times Id)\downarrow(Id\times Id\times P))) \\
= & \{\textit{relations } R\times Id\times Id \textit{ and } Id\times Id\times P \textit{ are orthogonal}\} \\
 & (A'\circ B')^n.\,(A'.\,(\mu(B\circ A)\downarrow(Id\times Id\times P)\downarrow(R\times Id\times Id))) \\
\sqsubseteq & \{\textit{definition of } B\overset{\mathcal{A}}{\sqsubseteq}_P B'\} \\
 & (A'\circ B')^n.\,(A'.\,((B'.\,(\mu(A\circ B)\downarrow(Id\times Id\times P)))\downarrow(R\times Id\times Id)))
\end{array}
$$

$$
\begin{array}{ll}
\sqsubseteq & \{\textit{coercion propagation lemma}\ \} \\
& (A' \circ B')^n.\,(A'.\,(B'.\,(\mu(A \circ B) \downarrow(Id \times Id \times P) \downarrow(R \times Id \times Id)))) \\
= & \{\textit{relations } Id \times Id \times P \textit{ and } R \times Id \times Id \textit{ are orthogonal}\} \\
& (A' \circ B')^n.\,(A'.\,(B'.\,(\mu(A \circ B) \downarrow(R \times Id \times P)))) \\
= & \{f^{n+1}.x \ = \ f^n.\,(f.x),\ \textit{definition of composition}\} \\
& (A' \circ B')^{n+1}.\,(\mu(A \circ B) \downarrow(R \times Id \times P))\ \Box
\end{array}
$$

Now using this lemma we can prove the second subgoal of the Modular Reasoning Theorem. Assume $A \overset{\mathcal{B}}{\sqsubseteq}_R A'$, $B \overset{\mathcal{A}}{\sqsubseteq}_P B'$, and $\forall Xb \bullet\ \mu(A' \circ B') = (A' \circ B')^n.\,Xb$, for fixed but arbitrary n. The conclusion is then proved as follows:

$$
\begin{array}{ll}
& \mu(A \circ B)\downarrow(R \times Id \times P) \\
\sqsubseteq & \{\textit{Lemma}\} \\
& (A' \circ B')^n.\,(\mu(A \circ B)\downarrow(R \times Id \times P)) \\
= & \{\textit{the third assumption, instantiating } Xb \textit{ with} \\
& \quad \mu(A \circ B)\downarrow(R \times Id \times P)\} \\
& \mu(A' \circ B')
\end{array}
$$

The proof of the third subgoal is similar. $\Box$

5.2 Interpretation and Implications of the Theorem

Let us consider how the requirement "no call-back assumptions" is reflected in the formulation of the theorem. In fact, this requirement is not captured by a separate assumption in the theorem, rather the definition of component refinement in context accommodates for it. As stipulated by this requirement, when refining the component $\mathcal{A}$ to $\mathcal{A}'$ we should not assume that the component $\mathcal{B}$ calls back methods of $\mathcal{A}'$, because in doing so we would implicitly modify the specification of the component system. The specification of method bodies of $\mathcal{A}$ is mathematically defined by $\mu(A \circ B)$, whereas the specification of method bodies of $\mathcal{B}$ is defined by $\mu(B \circ A)$. Accordingly, refinement between the specification of method bodies of $\mathcal{A}$ and the implementation of methods of $\mathcal{A}'$ in context of the specification of method bodies of $\mathcal{B}$ is expressed as follows:

$$(\mu(A \circ B))\downarrow(R \times Id \times Id) \ \sqsubseteq A'.\,(\mu(B \circ A)\downarrow(R \times Id \times Id))$$

Here the concrete coercions are necessary for adjusting the state spaces of the participating components. The same requirement for the refinement between B and B' in context of A is treated similarly.

Unlike in the case of "no call-back assumptions", the "no accidental mutual recursion" requirement is captured in the assumptions (c) and (d) of the theorem explicitly. Let us consider the assumption (c) (the assumption (d) is treated similarly):

$$(\exists n \bullet \forall Xb \bullet \mu(A' \circ B') = (A' \circ B')^n.Xb)$$

In this formula $(A' \circ B')^n$ is the function resulting from composing the function $(A' \circ B')$ with itself $n-1$ times. The intuition here is as follows. If the result of applying the function $(A' \circ B')$ to an arbitrary list of method bodies a finite number of times is equal to the complete unfolding of method invocations between $\mathcal{A}'$ and $\mathcal{B}'$, then the bodies of methods of A' are completely defined. This, of course, can only be achieved if the unfolding terminates, i.e. there is no infinite mutual recursion.

The "no accidental mutual recursion" requirement is non-modular in the sense that it requires checking for the absence of mutual recursion in the system composed from refined components. We envision several approaches to satisfying this requirement in a modular manner. For example, component methods in the original specification can be marked as atomic if they do not call other methods. While refining a component, atomic methods must remain atomic and non-atomic ones can introduce new calls only to atomic methods. Although being apparently restrictive, this approach guarantees the absence of accidental mutual recursion in the refined composed system. With another approach we can assign to every method an index which indicates the maximal depth of method calls that this method is allowed to make. This approach apparently only works if the original specification does not have mutually recursive method calls. For example, a method m which does not invoke any other method will have index 0, whereas a method n invoking m will have index 1. If a method invokes several methods with different indices, it is assigned the maximal of these indices plus one. With the original specification annotated in this manner we can require that, while refining a method, calls to methods with indices higher than the indices of the methods that were called before cannot be introduced. However, the detailed analysis of the different methods for establishing the "no accidental mutual recursion" requirement in a modular manner is outside the scope of this paper.

6 Discussion, Conclusions, and Related Work

We study a problem which hinders independent development of components in the presence of re-entrance. A formal analysis of this problem allowed us to recognise the essence of the problem in the conflict of assumptions that developers of components make about the behaviour of other components in the system.

Problems related to compositionality of systems have been and remain a subject of intensive studies in the formal methods community, e.g. [6].

In particular, compositionality of concurrently executing processes communicating through global variables have been the focus of formal analysis by Abadi and Lamport in [1]. The setting that they consider, although somewhat similar to ours, is more complicated. The results that Abadi and Lamport present are of very general applicability. As they study behaviour of concurrent processes in terms of the traces they leave on the externally observable variables, interpreting their results for the case of non-concurrent components communicating through method invocations is non-trivial.

Problems with re-entrance are also often discussed in the context of concurrent programming. In a multithreaded environment several instances of the same procedure modifying global variables can be executed simultaneously. One thread of control can enter the procedure and, before the end of the procedure is reached, a second thread of control can re-enter the same procedure. Apparently, such a situation is problematic because the second instance of the procedure might observe the global variables in an inconsistent state, or it can modify these global variables and then the first instance will observe them in an inconsistent state.

The problem that we consider is sufficiently different from the re-entrance problem as known in concurrent programming to deserve a separate name, the "component re-entrance problem". There are two scenarios in which this problem can occur; firstly, when components are independently developed from specifications and, secondly, during independent maintenance of components.

One of the recommendations in concurrent programming is to circumvent the re-entrance problem by avoiding the re-entrance setting, which can be achieved using various locking mechanisms. In object-oriented and component-based programming the re-entrance setting can be avoided by following what is known as the "push" communication style. Adhering to this communication style requires passing to a client component all the data it might possibly need as method parameters. Apparently, such an approach to component communication is rather inefficient, and it is often preferable to pass to the client component just a reference to itself and permit it to obtain all the data it might need. However, the latter approach, which is often referred to as the "pop" approach, matches the re-entrance setting.

Several researchers have pointed out that components should specify relevant information about their environments, such as required interfaces [13]. It was also recognised that accidental reuse does not lead to the development of robust maintainable systems [8]. To be really useful, reuse must be pre-planned by system developers. Agreeing with these ideas, we advocate a specification method where component environments are described by abstract specifications of their behaviour. We believe that the specification of the environment should be split into components specifying certain interfaces to indicate the communication protocol between the components. As the specifications of the environment components can be given in terms of abstract mathematical data structures and non-deterministic specification statements, this would permit a multitude of different implementations.

Similar problems occurring during maintenance of mutually dependent components have been mentioned by several researchers, e.g. Bertrand Meyer in [10] and Clemens Szyperski in [16]. Meyer considers the setting with two mutually dependent classes whose invariants include each other's attributes. His method for verification of conformance between two implementations of one class requires that the new implementation respect the invariant of the original implementation. He notices that this requirement alone is not sufficient for establishing correctness of the composed system and refers to this problem as "indirect invariant effect". He then makes the conjecture that mirroring such interclass invariants in the participating classes would be sufficient to avoid the problem. Although we disagree with the practice of stating interclass invariants, it appears that the problem considered by Meyer is just a special case of the component re-entrance problem as formulated in this paper. As our examples demonstrate, preserving invariants, taken alone, does not eliminate the problem.

Szyperski describes a similar problem but sees it rather as an instance of the re-entrance problem as occurring in concurrent systems. He reiterates the common recommendation for avoiding the problem, which suggests to establish a component invariant before invoking any external method. Interestingly enough, the recommendation to re-establish the invariant before all external method calls does not follow from the specification and is rather motivated by empirical expertise. As demonstrated by our examples, this recommendation, although being necessary, is insufficient.

In fact, our "no call back assumptions" requirement subsumes this recommendation. Let us reconsider our first example. According to the Modular Reasoning Theorem, to demonstrate that *Model′* is a valid implementation of *Model* in the context of *View*, we would need to show that every method of *Model′* calling methods of *View* composed with methods of *Model* refines the corresponding methods of *Model* composed with methods of *View*. Since *Model* and *Model′* operate on different attributes, to express, for example, in the method *append* of *Model′* the behaviour of a call to *View.update*, which calls *get_s* of *Model*, we need to coerce this call using an abstraction relation. Such an abstraction relation usually includes component invariants, and in this case includes the component invariant $n = \#s$ of *Model′*, i.e. $R.(s', n').s \;\widehat{=}\; s' = s \wedge n' = \#s'$. Note that in the definition of R the attributes of *Model′* are primed in order to distinguish them from the attributes of *Model*. According to the definition of refinement in context, the proof obligation for the method *append* after expansion and simplification is

$$(s := s \frown t; print(\#s)) \downarrow R \sqsubseteq s := s \frown t; (print(\#s)) \downarrow R; n := n + \#t$$

The right hand side can be expanded to $s := s\frown t; \{R\}; print(\#s); [R^{-1}]; n :=$

$n + \#t$. The abstraction statement preceding the invocation of *print* aborts, because it tries to find an abstract value of a sequence s satisfying the invariant $\#s = n$ which obviously does not hold at this point. Certainly, an aborting method is not a refinement of a non-aborting one and, therefore, $Model'$ fails to correctly implement *Model* in the context of *View*, breaching our requirement.

The requirement to re-establish a component invariant before all external calls is rather restrictive, because re-establishing the invariant might require a sequence of method calls to this and other components. Besides, it is not always necessary to establish the entire component invariant before external calls, because clients of the component can depend on some parts of the component invariant while being indifferent to the other parts. Szyperski in [16] proposes to "weaken invariants conditionally and make the conditions available to clients through test functions". In a way, he proposes to make assumptions that component developers make about other components more explicit. This idea can be elaborated through augmenting the specification of components with require/ensure statements stipulating assumptions and guarantees that the components make. To avoid a conflict of assumptions, the component specification can make explicit the information the component relies on and provides to other components. For instance, every method can begin with a require condition and end with an ensure condition. Also every method invocation can be surrounded by an ensure/require couple. Then, while implementing a method, the developer can assume the information as stipulated in the require condition and ought to establish the ensure condition. Such an explicit statement of mutual assumptions and guarantees between components would reduce the need to unfold method invocations when verifying refinement in context. Note that the theoretical underpinning of such an approach to specification of component systems is an interpretation of the results presented in this paper, as the refinement calculus includes constructs for expressing the require/ensure statements.

A specification and verification method for component systems based on such an approach should additionally provide for satisfying the "no accidental mutual recursion" requirement in a modular manner. The detailed elaboration of such a method represents the subject of current research.

As was already mentioned, we have made a number of simplifications in the component model. In particular, we have assumed that components do not have self-calls and component implementations do not introduce new methods. Relaxing these confinements on the component model is the subject of future work.

Acknowledgments

We would like to express our gratitude to Anna Mikhajlova for useful comments. Ralph Back and Joakim von Wright have provided valuable feedback on the earlier version of this paper. Discussions with Eric Hehner and his colleagues while presenting this work at the University of Toronto helped us to improve presentation of the material.

References

[1] M. Abadi and L. Lamport. Composing specifications. In *Stepwise Refinement of Distributed Systems*, number 430 in LNCS, pages 1–41, 1987.

[2] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[3] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, April 1998.

[4] M. Büchi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of WCOP'97*, volume 5 of *TUCS General Publication*, pages 23–32, June 1997.

[5] M. Büchi and W. Weck. A plea for grey-box components. Technical Report TUCS-TR-122, Turku Centre for Computer Science, Finland, Sept. 5, 1997.

[6] W.-P. de Roever, H. Langmaack, and A. Pnueli. *Compositionality: The Significant Difference. Proceedings of COMPOS'97*, volume 1536 of *LNCS*. Springer, 1997.

[7] J. Feiler and A. Meadow. *Essential OpenDoc*. Addison-Wesley, 1996.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 169–180, Oct. 1990.

[10] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[11] L. Mikhajlov, E. Sekerinski, and L. Laibinis. Developing components in the presence of re-entrance. Technical Report 239, Turku Centre for Computer Science, February 1999.

[12] C. C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.

[13] A. Olafsson and D. Bryan. On the need for "required interfaces" of components. In M. Muehlhaeuser, editor, *Special Issues in Object Oriented Programming*, pages 159–165. dpunkt Verlag Heidelberg, 1997. ISBN 3-920993-67-5.

[14] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[15] Sun Microsystems. *Java Beans(TM)*, July 1997. Graham Hamilton (ed.). Version 1.0.1.

[16] C. Szyperski. *Component Software – Beyond Object-Oriented Software*. Addison-Wesley, 1997.

[17] A. Tarski. A lattice theoretical fixed point theorem and its applications. *Pacific J. Mathematics*, 5:285–309, 1955.

[18] J. Wright. Program refinement by theorem prover. In *6th Refinement Workshop*, London, 1994. Springer–Verlag.

Appendix

Before proving the coercion propagation lemma we first need to consider the full language of the refinement calculus. It has the following basic statements and statement constructors:

$$S ::= [R] \mid \{R\} \mid S_1; S_2 \mid (\sqcap i \in I \bullet\ S_i) \mid (\sqcup i \in I \bullet\ S_i)$$

The demonic and angelic choice operators $\sqcap$ and $\sqcup$ are defined as follows:

$$\begin{array}{lcl} (\sqcap i \in I \bullet\ S_i).\,q & \widehat{=} & (\cap i \in I \bullet\ S_i.\,q) \\ (\sqcup i \in I \bullet\ S_i).\,q & \widehat{=} & (\cup i \in I \bullet\ S_i.\,q), \end{array}$$

where $\cap$ and $\cup$ respectively are the intersection and union of sets of states defined by the predicates $(S_i.\,q)$.

Furthermore, the language of the refinement calculus has derived statements that can be expressed using the basic statements in the following manner:

$\mathbf{skip} = \{Id\}$
$\mathbf{abort} = \{\lambda\sigma \bullet \lambda\gamma \bullet \mathbf{F}\}$, where $\mathbf{F}$ is boolean value false
$\mathbf{magic} = [\lambda\sigma \bullet \lambda\gamma \bullet \mathbf{F}]$
$[p] = [\lambda\sigma \bullet \lambda\gamma \bullet (p.\,\sigma)\ \wedge\ (\sigma = \gamma)]$
$\{p\} = \{\lambda\sigma \bullet \lambda\gamma \bullet (p.\,\sigma)\ \wedge\ (\sigma = \gamma)\}$
$\mathbf{if}\ g\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} = ([g]; S_1) \sqcap ([\mathbf{not}\ g]; S_2)$
$\mathbf{while}\ g\ \mathbf{do}\ S\ \mathbf{od} = (\mu X \bullet [g]; S; X \sqcap [\mathbf{not}\ g])$
$|[\mathbf{var}\ x \mid p \bullet S\ ]| =$
$\quad [\lambda y \bullet \lambda(x', y') \bullet (p.\,(x', y'))\ \wedge\ (y' = y)]; S; [\lambda(x, y) \bullet \lambda y' \bullet (y' = y)]$

Here $[p]$ is the *assumption* statement, which behaves as **skip** if p holds and as **magic** otherwise. The statement $|[\mathbf{var}\ x \mid p \bullet S\ ]|$ is a block statement introducing a local variable x and initializing it according to the predicate p. The statements **if** and **while** have the usual interpretations.

Bodies of component methods are composed of the refinement calculus statements. In our customised formalisation of components we model method parameters by means of global variables. Thus methods of the composed system $\mathcal{A}\,\mathbf{comp}\,\mathcal{B}$ operate on the state space $\Sigma \times \Delta \times \Gamma$ consisting of the internal state space of the component $\mathcal{A}$, parameters of methods of $\mathcal{A}$ and $\mathcal{B}$, and the internal state space of $\mathcal{B}$. To model data encapsulation, we permit the statements in methods of $\mathcal{A}$ operate only on $\Sigma \times \Delta$, and we let methods of $\mathcal{A}$ to access the state of $\mathcal{B}$ only by invoking its methods. In our customised theory of components method invocation is modelled by syntactic substitution of the body of the called method in a calling method. To distinguish the statement operating on the other part of the state space, we introduce the statement **call** m_i which is just a syntactic sugar for the statement representing the body of the method m_i of the component $\mathcal{B}$.

Accordingly, we have that all statements, except **call** m_i, in methods of $\mathcal{A}$ have the structure $S \times$ **skip**, where **skip** operates on the state space Γ. The statements **call** m_i have the structure **skip** $\times S$. The dual layout holds for the methods of $\mathcal{B}$.

We use the following rules for calculating data refinements of the refinement calculus statements:

$$\begin{array}{ll}
(a) & (R^{-1};Q' \subseteq Q;R^{-1}) \Rightarrow [Q]{\downarrow}R \sqsubseteq [Q'], \\
(b) & (R;Q \subseteq Q';R) \Rightarrow \{Q\}{\downarrow}R \sqsubseteq \{Q'\}, \\
(c) & (S_1; S_2){\downarrow}R \sqsubseteq (S_1{\downarrow}R); (S_2{\downarrow}R), \\
(d) & (\sqcap i \in I \bullet S_i){\downarrow}R \sqsubseteq (\sqcap i \in I \bullet S_i{\downarrow}R), \\
(e) & (\sqcup i \in I \bullet S_i){\downarrow}R = (\sqcup i \in I \bullet S_i{\downarrow}R)
\end{array}$$

Proofs of these rules are due to Ralph Back and Joakim von Wright.

Coercion Propagation Lemma. *For functions*

$$\begin{array}{l}
A : \mathcal{L}_n(Ptran(\Sigma \times \beta)) \to \mathcal{L}_m(Ptran(\Sigma \times \beta)), \\
B : \mathcal{L}_m(Ptran(\alpha \times \Gamma)) \to \mathcal{L}_n(Ptran(\alpha \times \Gamma)),
\end{array}$$

tuples of statements

$$Xb : \mathcal{L}_n(Ptran(\Sigma \times \Gamma)),\ Yb : \mathcal{L}_m(Ptran(\Sigma \times \Gamma)),$$

and relations $P : \Gamma' \leftrightarrow \Gamma$, $R : \Sigma' \leftrightarrow \Sigma$ the following holds:

$$\begin{array}{lcl}
(A.\,Xb){\downarrow}(Id \times P) & \sqsubseteq & A.\,(Xb{\downarrow}(Id \times P)) \\
(B.\,Yb){\downarrow}(R \times Id) & \sqsubseteq & B.\,(Yb{\downarrow}(R \times Id))
\end{array}$$

Proof We prove the first part of the lemma, the proof of the second part is similar. To make instantiations of type variables explicit, we write A_Γ for the function A with polymorphic type variable β instantiated with Γ. Similarly, A'_Γ stands for A with β instantiated with Γ'. Now we can rewrite our goal as follows:

$$(A_\Gamma.\,Xb){\downarrow}(Id \times P) \sqsubseteq A_{\Gamma'}.\,(Xb{\downarrow}(Id \times P))$$

Applying the function A to the tuple of statements Xb results in a tuple of the refinement calculus statements, as described above. Without loss of generality we can consider only the case when the resulting tuple is singleton. We prove the coercion propagation lemma by induction on the statements of the refinement calculus.
Case 1. $A.\,Xb = [R \times Id]$

$$\begin{array}{ll}
 & (A_\Gamma.\,Xb){\downarrow}(Id \times P) \\
= & [R \times Id]{\downarrow}(Id \times P)
\end{array}$$

$$
\begin{array}{ll}
\sqsubseteq & \{\textit{rule (a), definition of cross product and} \\
& \quad \textit{composition on relations}\ \} \\
& [R \times Id] \\
= & \{[R \times Id] \textit{ does not depend on } Xb\} \\
& A_{\Gamma'}.\,(Xb \downarrow (Id \times P))
\end{array}
$$

Case 2. $A_{\Gamma}.\,Xb = \{R \times Id\}$

$$
\begin{array}{ll}
& (A_{\Gamma}.\,Xb) \downarrow (Id \times P) \\
= & \{R \times Id\} \downarrow (Id \times P) \\
\sqsubseteq & \{\textit{rule (b), definition of cross product and} \\
& \quad \textit{composition on relations}\ \} \\
& \{R \times Id\} \\
= & \{\{R \times Id\} \textit{ does not depend on } Xb\} \\
& A_{\Gamma'}.\,(Xb \downarrow (Id \times P))
\end{array}
$$

Case 3. $A_{\Gamma}.\,Xb = (A_{\Gamma 1}.\,Xb);(A_{\Gamma 2}.\,Xb)$
For this case we have an inductive assumption of the form:

$$
\begin{array}{l}
(A_{\Gamma 1}.\,Xb) \downarrow (Id \times P) \sqsubseteq A_{\Gamma' 1}.\,(Xb \downarrow (Id \times P)) \\
(A_{\Gamma 2}.\,Xb) \downarrow (Id \times P) \sqsubseteq A_{\Gamma' 2}.\,(Xb \downarrow (Id \times P))
\end{array}
$$

$$
\begin{array}{ll}
& (A_{\Gamma}.\,Xb) \downarrow (Id \times P) \\
= & (A_{\Gamma 1}.\,Xb; A_{\Gamma 2}.\,Xb) \downarrow (Id \times P) \\
\sqsubseteq & \{\textit{rule (c)}\ \} \\
& (A_{\Gamma 1}.\,Xb) \downarrow (Id \times P); (A_{\Gamma 2}.\,Xb) \downarrow (Id \times P) \\
\sqsubseteq & \{\textit{inductive assumptions}\ \} \\
& (A_{\Gamma' 1}.\,(Xb \downarrow (Id \times P))); (A_{\Gamma' 2}.\,(Xb) \downarrow (Id \times P)) \\
= & A_{\Gamma'}.\,(Xb \downarrow (Id \times P))
\end{array}
$$

Case 4. $A_{\Gamma}.\,Xb = (\sqcap i \in I \bullet\ A_{\Gamma i}.\,Xb)$
For this case we have an inductive assumption of the form:

$$
\forall i \in I \bullet\ (A_{\Gamma i}.\,Xb) \downarrow (Id \times P) \sqsubseteq A_{\Gamma' i}.\,(Xb \downarrow (Id \times P))
$$

$$
\begin{array}{ll}
& (A_{\Gamma}.\,Xb) \downarrow (Id \times P) \\
= & (\sqcap i \in I \bullet\ A_{\Gamma i}.\,Xb) \downarrow (Id \times P) \\
\sqsubseteq & \{\textit{rule (d)}\ \} \\
& \sqcap i \in I \bullet\ (A_{\Gamma i}.\,Xb) \downarrow (Id \times P) \\
\sqsubseteq & \{\textit{inductive assumption}\ \} \\
& \sqcap i \in I \bullet\ A_{\Gamma' i}.\,(Xb \downarrow (Id \times P)) \\
= & A_{\Gamma'}.\,(Xb \downarrow (Id \times P))
\end{array}
$$

Case 5. Proof is similar to the proof of case 4.

Case 6. $A_\Gamma . Xb = \textbf{call}\ m_i$, where $m_i \in Xb$

$$
\begin{aligned}
& (A_\Gamma . Xb)\downarrow(Id \times P) \\
= \quad & (\textbf{call}\ m_i)\downarrow(Id \times P) \\
\sqsubseteq \quad & \{\textit{definition of}\ \textbf{call}\ \} \\
& m_i\downarrow(Id \times P) \\
= \quad & \textbf{call}\ (m_i\downarrow(Id \times P)) \\
= \quad & A_{\Gamma'} . (Xb\downarrow(Id \times P))
\end{aligned}
$$

The proofs of derived statement cases are similar to those given above. $\Box$

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**