Verifying Reuse Contracts with a Component Model

Agustín Cernuda del Río Jose Emilio Labra Gayo Juan Manuel Cueva Lovelle

Departamento de Informática
Universidad de Oviedo
C/Calvo Sotelo S/N
33007 Oviedo, Spain
+34 985 10 {50 94, 33 94, 33 96}
{guti, labra, cueva}@lsi.uniovi.es

Abstract

The Itacio component model intends to bring a method of verifying software systems made up of components. This method can be applied at different levels of abstraction, and to different facets of the software development process. All that is needed is to define the correspondence between the Itacio model basic entities –components, sources, sinks- and the entities of the addressed problem.

As an example, one of the potential uses of this model is the verification of system evolution. The interaction pattern between several objects can be described as a so-called reuse contract; the potential modifications of this contract have already been formalised as operators. Considering these contracts and operators as Itacio components, the evolution of a system can be modelled as their composition. This complex can be verified by the usual Itacio inference process, and inconsistencies or undesired effects in the evolution of a system can be foreseen.

The main advantages of Itacio is that it brings a very simple model that can be applied in the real world without the need of a very specialised training, and the deliberate generality of this model allows it to be applied to different kinds of software structures. The time evolution modelled and presented in this article is a good example, since the Itacio model was not designed with this specific use in mind.

KEYWORDS: software components, component model, verification, evolution, reuse, reuse contracts.

1. Introduction

Building software upon components allows to reduce development costs and shortening time-to-market. But current component technologies (COM, CORBA, JavaBeans) usually solve cross-platform or low-level technical problems. Automatic composition checking is usually restricted to signature matching; in addition, the evolution and maintenance of systems can lead to problems as active restrictions are unintentionally violated.

We think that components should explicitly incorporate assumptions –about what a component produces and also about what it expects from its neighbours- in a manner that is statically, automatically verifiable, and that the component concept can offer a reasoning framework that can be applied much further than at the “library” level, as it is frequently used nowadays. For instance, it can be used to keep trace of (and verify) restrictions as systems evolve.

In this paper, the Itacio component model is briefly introduced. Then, some previous work in the area of reuse contracts and operators is summarised. Afterwards, the application of this model to reuse contracts is described and an example of use is presented. Finally, conclusions are obtained from these experiences.

2. The Itacio Component Model

The Itacio component model [1] offers a way of verifying software systems built by joining components. The main advantages of this model are that no execution of the program is needed, and the influence of components (even indirect or side-effect influence) is easily taken into account without data flow analysis. Also, the specification system is fully modular and, in addition, it can be easily supported by a Constraint Logic Programming system. Finally, this model can be applied at different levels of abstraction. The model deliberately avoids to bind the user to a specific semantic notion of component, so that he or she can apply a general verification framework to a very wide spectrum of problems.

A precise description of this model can be found in [2]. The central idea of the Itacio model is a flexible definition of a component. A *component* C is an entity which has a *frontier* F(C) and a set of *restrictive expressions* E(C).

F(C) is a finite set whose elements are called *connection points*; these connection points can be *sources* (whose set is denoted by S(C)) or *sinks* (whose set is denoted by K(C)). Informally stated, sources carry information outside of a component (e.g., a function call) and sinks introduce information into a component (e.g., a function's entry point).

Restrictive expressions are also divided into two disjoint subsets. The set of *requirements* R(C) contains restrictive expressions that are Horn clauses (a special form of first-order logic predicates) over the sinks. The set of *guarantees* G(C) contains Horn clauses over both sinks and sources. In addition, there is a one-to-one correspondence between the sinks and the requirements (there is one requirement predicate associated to each sink, although this predicate can refer to more than one sink). Requirements do not refer to sources because this system intends to verify the *composition* of components, not the internal behaviour of a component; so it is assumed that we have control over the behaviour of the component itself and we do not need to restrict our own outputs. Maybe another component will (in its restrictions over its own inputs or sinks).

A *system* $\Omega = \{\nu, \varepsilon, \boldsymbol{L}\}$ is a finite graph whose nodes ν are components and whose edges ε are source/sink pairs, together with a set $\boldsymbol{L}$ of auxiliary predicates called the *library*. In other words, a system is built by taking components and connecting each and every source with some sink, and adding some auxiliary predicates. The first requirement for a system (the so-called *topological correctness*) establishes that there will be no isolated connection points and each one of them will be connected only once. These topological correctness restrictions exist only for prototype simplicity reasons, but in a very near future the model will be extended to handle the connection of a source with multiple sinks or vice versa.

We then define the *raw knowledge base* $\boldsymbol{K}_r(\Omega) = \{p \,/\, (p \in R(C) \vee p \in G(C)), C \in \nu\}$; it is the concatenation of all the restrictive expressions of the components of the system.

From its definition, it can be seen that $\boldsymbol{K}_r(\Omega)$ does not depend on ε; so it does not contain any information about connections. The *knowledge base* $\boldsymbol{K}(\Omega)$ is built by taking $\boldsymbol{K}_r(\Omega)$ and following an iterative substitution process (detailed in [2]) over all the source and sink names so that, if some $s_i \in S(C_m)$ and $k_j \in K(C_n)$ are connected, a new, unique atom name a is generated, and all the occurrences of either s_i or k_i in $\boldsymbol{K}_r(\Omega)$ are substituted by a. The *knowledge base* $\boldsymbol{K}(\Omega)$ resulting from this process implicitly contains the information about the topology of the system. The building process also ensures that the relationship between each resulting requirement and its associated sink is not lost.

Finally, the verification model relies on an inference process over $\boldsymbol{K}(\Omega)$. The system is considered to be correct if each and every restrictive expression of $\boldsymbol{K}(\Omega)$ is proven to be true. Also, since each requirement in $\boldsymbol{K}(\Omega)$ is related to one sink, if some restriction is not fulfilled it is possible to know exactly which connection point is failing and why.

A first prototype for this model was implemented with a diagramming tool [1]; after that, a Java/XML/VML prototype (with a web-based user interface) was built [2]. Experiments with this prototype have allowed refining the structure of the model and testing its application to new abstraction layers of the development process to verify that this simple schema is useful and will behave as expected in different situations. One of the explored areas (with a good response of the model, which did not require any adaptation at all even when this use had not been anticipated) is reuse contracts.

3. Reuse Contracts

The idea of contracts applied to software has been explored under different interpretations (see [4], [6], [7]). In this paper, the "reuse contract" approach due to Carine Lucas [6] is examined. First, because it is a static, declarative model, which better fits our ideas (static analysis is one of the key points for us). Moreover, Lucas contract model allows a higher modularisation than others, so that it is easier to describe in Itacio. So, from now on, the term "contracts" will refer to the reuse contracts formalized by Lucas. Part of her work will be summarised next.

Reuse contract: A *reuse contract* is a set of participants, each with a name (unique within this reuse contract), an acquaintance clause and an interface.

Acquaintance clause: If a participant in a contract "knows about" other participant, the former will have an *acquaintance clause* where the latter is mentioned. The calling participant will refer to the called one by means of an acquaintance name, a sort of alias which is also stated in the acquaintance clause.

Interface: The interface of a participant is basically a set of *operation*[1] descriptions. Each description contains an operation name and a list of the operations it calls; these are qualified with the acquaintance name.

With these elements, the collaboration between a set of participants can be modelled. A reuse contract contains information about the participants, the operations they implement, the participants they need to know about for that reason, and the dependencies between operations.

It is possible to build an inconsistent reuse contract; for instance, operations can refer to non-existing operations or participants. To avoid this, correction criteria are defined; a contract is said to be **well-formed** if it satisfies three basic well-formedness clauses, WF1 / WF2 / WF3. WF1 requires that acquaintance clauses refer to existing participants, WF2 requires that operation dependencies refer to valid acquaintances, and WF3 requires that the invoked operation exists in the referenced participant. Lucas original work does not state more clauses because her notation implicitly avoids any other error but, if using a different notation, additional clauses may be necessary (this is our case).

Apart from the reuse contract itself, Lucas includes a study of the modifications that a contract can suffer. These modifications adopt the form of **operators**. They are grouped upon scope and kind of modification.

On one hand, there are:

- **Participant** modifications, when the modifications affect the operations of participants (scope of the modification: inside the participant).
- **Context** modifications, when the modifications affect the existence of participants or the relationship between participants (scope of the modification: outside a participant).

On the other hand, there are:

- **Extension** modifications (adding elements).
- **Cancellation** modifications (removing elements).
- **Refinement** modifications (adding dependencies between elements).
- **Coarsening** modifications (removing dependencies between elements).

Since these two modification groups are orthogonal, the result is eight possible combinations, and hence eight operators.

- **Participant extension**: adding operations to existing participants.
- **Context extension**: adding participants to a contract.

[1] Be careful not to confuse *operation* (something similar to *method* in OOP) with contract *operator* (explained later).

- **Participant cancellation**: removing operations from a participant.
- **Context cancellation**: removing participants from a contract.
- **Participant refinement**: adding dependencies (calls) between operations of participants.
- **Context refinement**: adding dependencies between participants (acquaintance clauses).
- **Participant coarsening**: removing dependencies (calls) between operations of participants.
- **Context coarsening**: removing dependencies between participants (acquaintance clauses).

These operators are formally defined in [6]. First, the structure of their parameters is clearly stated; an operator takes a contract and a contract *modifier* (a list of participants, operations and so on) and produces a new resulting contract. The correctness criteria for operators involve the modifier and the contract it applies to; that is, the *applicability* of the operator. It is not enough for a modifier to be only morphologically correct; when it is applied to some contract by means of an operator, additional verifications must be made. For instance, a participant extension modifier could try to add an operation to a participant that does not exist in the target contract.

It must be noted that contract modifiers have a shape very similar to a contract (or to a contract fragment, in some cases). This is normal, since the structure of the information contained in a contract modifier is basically the same as in a contract, and in addition using a similar notation makes the implementation of prototypes much easier.

The eight reuse operators are *atomic*, in the sense that they collect the basic modifications a contract can suffer, and they are fully *modular*, in the sense that each one of them can be independently verified and applied. In addition, if an operator is correctly applied to a well-formed contract, it has been proven that the resulting contract will still be well-formed.

The drawback of this schema is that the use of only one operator would represent a very limited change to the system. Modifying a system usually involves the application of several operators, and there exists the concept of combined operators; if they are not used (this is our case) the maintenance action must be first decomposed into the atomic operators.

A chain of modifications to a contract can lead to errors or inconsistencies as the system evolves. Lucas proposes comparing the applied operators before applying them, using several “rules of thumb” to discard dangerous combinations. But the combination of reuse contracts and operators can also be addressed under the Itacio approach without the need of such rules. The following section will explain this.

4. Reuse Contracts Considered as Itacio Components

As said above, reuse contracts are basically descriptions of the interaction pattern between several participants. They collect information about the operations each participant offers, the acquaintance relationships between participants, and the operations invoked by each operation of a participant, be it inside the same participant or using the acquaintance relationship to call operations from other participants.

Participants/operations are much like objects/interfaces; it would be obvious to identify a component with an object (and hence with a participant). This is an old issue, revisited from time to time [3]. But this is not the approach of this paper (although Itacio can perfectly be applied under that interpretation; in fact, this was the initial motivation of this work).

Here, the Itacio component model is being applied in order to verify system evolution, so the notion of component is applied at a higher level of abstraction. Each contract (the whole contract, including all participants) will be considered a component. Also, a modifier (a “contract fragment” whose mission is to be used as a parameter in reuse contract operators) will be considered a component too.

A reuse contract fulfils the requirements for being considered a component [2]: it has a clear frontier and contains a set of restrictive expressions. This frontier will be considered to have only one source: the contract name, which allows recovering any information about the contract. It can be seen as something very similar to an “integer value” component; it would have only one source that gives an integer value. In this case, the only difference is that “a contract” is a much more complex value or data type than “an integer”, and represents a much higher level of abstraction; but the general role of both components is essentially the same.

Contracts (and contract fragments or modifiers) are not the only kind of component to be considered. Contract operators are also considered as components. A contract operator has also only one source (the contract that results from applying the operator) and two sinks: the original contract and the modifier contract. Once again, consider the example of an “Add” component; it would receive two integers and produce a new integer. Contract operator components receive and produce values that are more complex, but the structure of these components is essentially the same.

Contract: smplDrive	

PARTICIPANT: smplDriver
OPERATIONS:
::go()
    myCar::startEngine()
    myCar::pushGasPedal()
::stop()
    myCar::pushBrake()
    myCar::stopEngine()
::goFaster()
    myCar::pushGasPedal()
::goSlower()
    myCar::pushBrake()
	PARTICIPANT: smplCar
	OPERATIONS:
	::startEngine()
	::stopEngine()
	::pushBrake()
	::pushGasPedal()

ACQUAINTANCES: smplDriver::myCar->smplCar	

Fig. 1. An elementary sample contract

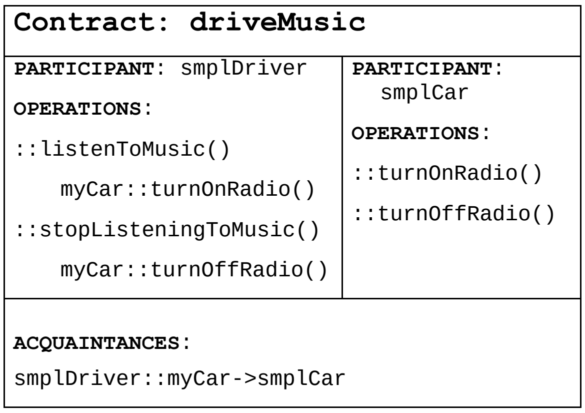
```
Type=smplDrive

Sources=res

BEGIN_RESTRICTIONS

isContract($res$).


participant($res$, smplDriver).
participant($res$, smplCar).

acqRelationship($res$, smplDriver, myCar,
   smplCar).

operation($res$, smplDriver, go).
operation($res$, smplDriver, stop).
operation($res$, smplDriver, goFaster).
operation($res$, smplDriver, goSlower).

operation($res$, smplCar, startEngine).
operation($res$, smplCar, stopEngine).
operation($res$, smplCar, pushBrake).
operation($res$, smplCar, pushGasPedal).


operationInvocation($res$, smplDriver, go,
   myCar, startEngine).
operationInvocation($res$, smplDriver, go,
   myCar, pushGasPedal).
operationInvocation($res$, smplDriver,
   stop, myCar, pushBrake).
operationInvocation($res$, smplDriver,
   stop, myCar, stopEngine).
operationInvocation($res$, smplDriver,
   goFaster, myCar, pushGasPedal).
operationInvocation($res$, smplDriver,
   goSlower, myCar, pushBrake).

END_RESTRICTIONS
```

Fig. 2. Contract of Fig. 1 expressed in Itacio prototype.

Given these component definitions, it is possible to model the evolution of a system as the application of a chain of operators over contracts. For instance, let us have an interaction pattern between a car and its driver. This scenario could be described with a reuse contract as described in Fig. 1. If an operation invokes some other operation, the invoked one appears indented. For example, `smplDriver::go()` invokes two operations of `smplCar`: `startEngine()` and `pushGasPedal()`. Also, `smplDriver` has an acquaintance clause in which it “knows about” `smplCar` and calls it `myCar`.

All of this information can be represented in a Constraint Logic Programming system with a few declarative statements (Fig. 2); in the Itacio prototype, a proper predicate structure has been designed to do this in an easy and readable manner. Also, a basic library for handling, querying and verifying these structures (the ***L*** mentioned in chapter 2) has been developed.

If this system must be modified so that the car has a radio and the driver is able to listen to music, a contract modifier could be built that adds the necessary operations to the involved participants, as described in Fig. 3.



Contract: driveMusic

PARTICIPANT: smplDriver	**PARTICIPANT**: smplCar
OPERATIONS:	**OPERATIONS**:
::listenToMusic()	::turnOnRadio()
myCar::turnOnRadio()	::turnOffRadio()
::stopListeningToMusic()	
myCar::turnOffRadio()	

ACQUAINTANCES:
smplDriver::myCar->smplCar

Fig. 3. A contract modifier to add operations to participants.

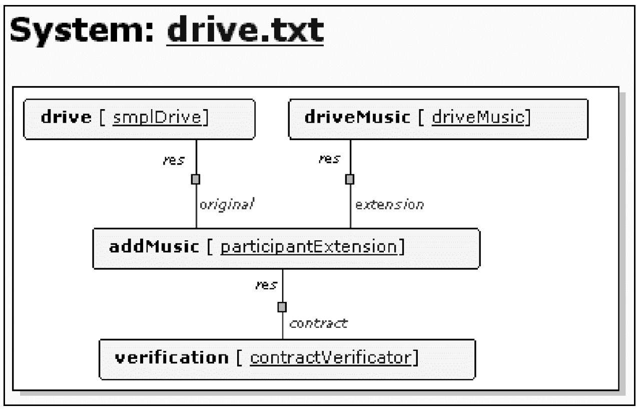


Fig. 4. A simple contract modification (Itacio prototype screen capture)

In order to combine these two contracts and bring a new contract that reflects the modification of the system, a contract operator component must be used. In this case, the involved operator is a *participant extension*. For any operator to be applied, the contract and the modifier must fulfil several criteria; the restrictive expressions of the operator component require this.

Finally, the resulting contract must be verified. A new component comes into action, with only one sink for the purpose of verification. This component has only one restriction: the contract it receives must be well-formed.

Combining all these components in Itacio, the result is a graph that models the evolution of the contract (Fig. 4). In this case, the combination of components is correct (to put it in reuse contracts terminology, contract `smplDrive` is well-formed, and it is *participant-extendible* by `driveMusic`).

5. Case Study: Saving Documents in MFC

The *Microsoft Foundation Classes*, or MFC for short [8], is a big set of classes written in C++. It is not only a mere class library, but a whole framework for developing applications for the Microsoft Windows operating systems family. The development tools from Microsoft generate application skeletons in MFC that the programmer can leverage. Windows native development in C/C++ requires quite a big amount of skeleton code, so using MFC helps productivity. The drawback is the same as in many other application development frameworks; its learning curve (as said above, MFC is really big) and the dependence on third-party (in this case Microsoft) code.

The example presented here has been extracted from a real case; the adopted solution was not a desirable one, but there were not many other options at that moment (and, anyway, it serves as an example of other different and more *normal* situations). The scenario was the development of a client/server graphical editor. One of the requirements for this system called for a modification of the document saving schema; when the user pushed the *Save* button of the editor, some specific things had to be done. As usual in application frameworks, the programmer had to modify the behaviour of some classes (by inheritance in this case), taking into account the MFC code environment. The original saving schema of MFC is reproduced in Fig. 5.

The outer rectangles represent the involved classes[2]. The inner rectangles represent operations (class methods), and the arrows represent the calling structure.

The operations that appear in grey, together with the thick arrows, represent the calling sequence that had to be modified. It was necessary to derive a new `CDocument` subclass (in terms of C++ programming), and implement a new version for the operation `OnSaveDocument()`. There was

[2] Actually, this schema has been a bit simplified. There were more than two classes involved, but the simplification made does not spoil the example and it gets much simpler and readable.

additional work to do; the behaviour implemented in `DoFileSave()` and `DoSave()` was not appropriate. The desirable solution would have been to redefine also these functions in the derived class; but it was not possible, because the design of the class library did not consider these function modifiable and they were not declared as *virtual*. This is a very common problem when using frameworks; their designers cannot anticipate all the possible uses or modifications to the original structure, and in addition the users must stick to an existing design, even when it is not the best one for their current purpose. They do not have the possibility of refactoring the code.

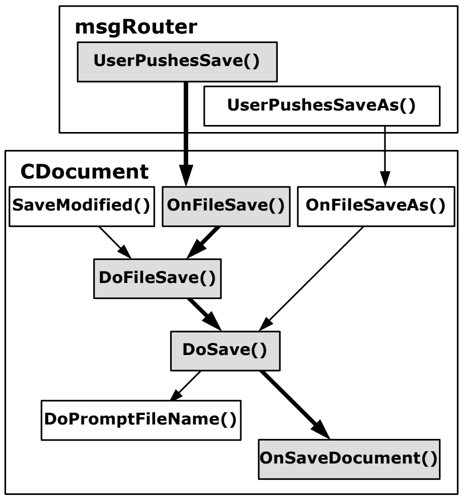


Fig. 5. Original calling structure for saving documents in an MFC application

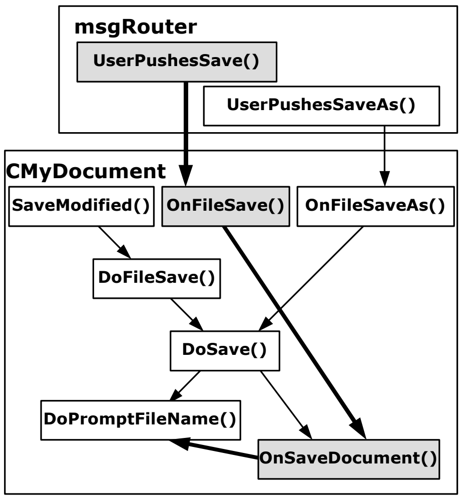


Fig. 6. Modified calling structure in CMyDocument

So the programmer decided to adopt an alternative approach. It was necessary to make a sort of bypass, so that `OnFileSave()`(which indeed happened to be declared as *virtual* and hence modifiable in the derived class) called `OnSaveDocument()` directly, avoiding the undesired effects of `DoFileSave()` and `DoSave()`. In order to substitute these functions, `OnSaveDocument()` had to rely (just as `DoSave()` did) on `DoPromptFilename()`, introducing a new dependency. This modified structure is depicted in Fig. 6.

Obviously, the described design involves its risks. The programmer has been forced to introduce dependencies that did not exist in the original design of the framework, and to alter its usual behaviour in an unpredicted way. The evolution and maintenance of this system is then compromised. Even if the assumptions the programmer made were carefully collected in some design document, this information would not automatically checkable.

This is a case where applying Itacio under the “reuse contracts” interpretation can be of value. First, the original structure of the MFC document saving system can be documented with a reuse contract that describes the interaction between the participants; see Fig. 7 for the contract schema and Fig. 8 to see how it appears expressed in Itacio.

With this starting point, the adaptations made to this schema can be also represented as contract operators. In this example, operators are applied “as is”, although frequently used operators can be combined to have pre-built operator chains packed as a single component. As initially stated in Lucas’ work, the combination of several operators must be evaluated under some combinability rules in order to predict the appearance of malformed contracts; but the Itacio inference model offers a significant advantage. It verifies this system using all its information [2] and specific “rules of thumb” are not necessary. A fragment of the system in an Itacio prototype can be seen in Fig. 9.

```
PARTICIPANT: cDocument
OPERATIONS:
::doFileSave()
    cDocument::doSave()
::doPromptFileName()
::doSave()
    cDocument::doPromptFileName()
    cDocument::onSaveDocument()
::onFileSave()
    cDocument::doFileSave()
::onFileSaveAs()
    cDocument::doSave()
::onSaveDocument()
::saveModified()
    cDocument::doFileSave()
```

```
PARTICIPANT: msgRouter
OPERATIONS:
::userPushesSave()
    cDocument::onFileSave()
::userPushesSaveAs()
    cDocument::onFileSaveAs()

ACQUAINTANCES:
cDocument::me->cDocument
   msgRouter::targetDoc->cDocument
```

Fig. 7. Original MFC document saving schema described as a reuse contract

```
Type=saveMFC
Sources=res
BEGIN_RESTRICTIONS

isContract($res$).

participant($res$, msgRouter).
participant($res$, cDocument).

acqRelationship($res$, msgRouter, targetDoc,
     cDocument).
acqRelationship($res$, cDocument, me,
     cDocument).

operation($res$, cDocument, onFileSave).
operation($res$, cDocument, onFileSaveAs).
operation($res$, cDocument, saveModified).
operation($res$, cDocument, doFileSave).
operation($res$, cDocument, doSave).
operation($res$, cDocument, doPromptFileName).
operation($res$, cDocument, onSaveDocument).

operation($res$, msgRouter, userPushesSave).
operation($res$, msgRouter,
     userPushesSaveAs).

operationInvocation($res$, msgRouter,
     userPushesSave, targetDoc, onFileSave).
operationInvocation($res$, msgRouter,
     userPushesSaveAs, targetDoc,
     onFileSaveAs).

operationInvocation($res$, cDocument,
     saveModified, me, doFileSave).
operationInvocation($res$, cDocument,
     onFileSave, me, doFileSave).
operationInvocation($res$, cDocument,
     doFileSave, me, doSave).
operationInvocation($res$, cDocument,
     onFileSaveAs, me, doSave).
operationInvocation($res$, cDocument, doSave,
     me, doPromptFileName).
operationInvocation($res$, cDocument, doSave,
     me, onSaveDocument).

END_RESTRICTIONS
```

Fig. 8. Contract of Fig. 7 expressed as an Itacio component

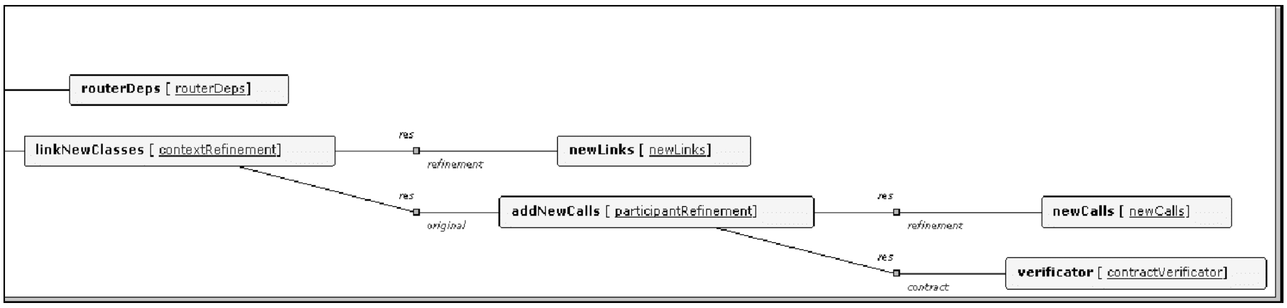
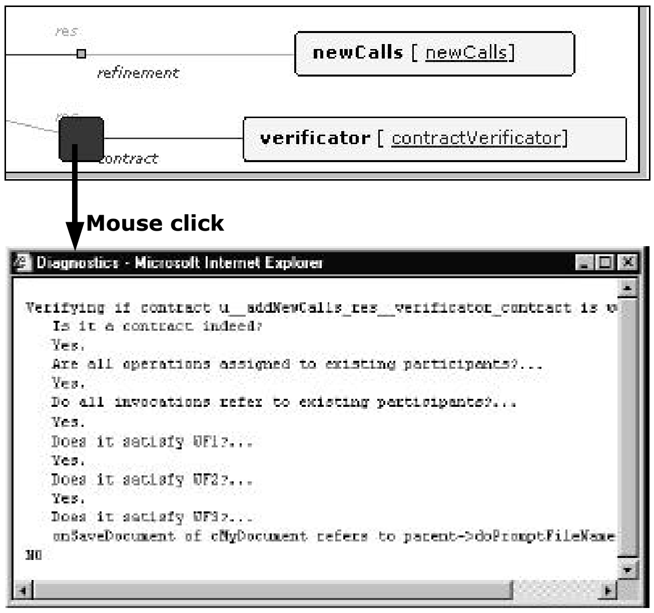


Fig. 9. MFC document saving system modeled in Itacio/Contracts (fragment)

The upper left box in this diagram (not shown) would be the contract that describes the original MFC behaviour, and the other boxes represent the modification chain until the desired new structure is achieved (bottom right). As said above, there could be a much smaller number of boxes if combined operators were used, but in this case we stick to the atomic, basic operators.

This structure represents, in fact, a knowledge base that can be used to make reasonings about the system. For instance, the next version of MFC could be modified so that the operation `DoPromptFileName()` (see Fig. 6) is removed. To evaluate how this change would impact the system, it is not necessary to make a complete, manual review. Simply, an operator is applied to the base MFC contract to represent the new behaviour (or the base contract is directly modified if the user does not want to explicitly model MFC evolution). Although the modification is made at the beginning of the chain, the system pinpoints the problem (with a big square) where it manifests: in this case, the result of the modification is a contract which is not well-formed (Fig. 10).

The system is able to explain the problem. In this case, the described change in the MFC implementation would break our code, because we made some assumptions when it was built. These assumptions can be violated when modifying MFC itself. Documenting these assumptions with reuse contracts offers a good way of detecting this kind of problems in a general way.


res
refinement
newCalls [newCalls]
res
verificator [contractVerificator]
contract
Mouse click
Diagnostics - Microsoft Internet Explorer

```
Verifying if contract u__addNewCalls_res__verificator_contract is w
   Is it a contract indeed?
   Yes.
   Are all operations assigned to existing participants?...
   Yes.
   Do all invocations refer to existing participants?...
   Yes.
   Does it satisfy WF1?...
   Yes.
   Does it satisfy WF2?...
   Yes.
   Does it satisfy WF3?...
   onSaveDocument of cMyDocument refers to parent->doPromptFileName
NO
```

Fig. 10. Detecting an inconsistency in contract evolution

The removal of a base class operation seems a radical modification, but it is not the only one to be detected. In fact, the Itacio model is extremely flexible, so potentially any restriction that can be expressed in terms of first-order logic predicates can be taken into account in the verification process.

For instance, let us suppose that the operation DoPromptFileName() (see Fig. 5 again) is not removed in the next version of MFC, but DoSave() stops calling it. Since our modified version of OnSaveDocument() is trying to mimic the behavior of DoSave() (and in fact this is the reason why OnSaveDocument() calls DoPromptFileName()) , if DoSave() stops making this call, that would be a very important information. This change would not break the basic contract structure, but it would break an important assumption.

Although the original contract model is not designed to express a restriction like “This contract requires that in the base contract operation A calls operation B”, it can be directly expressed in Itacio with only one declarative statement, with no specific adaptation of the system. If the operation invocation is removed from the MFC base contract, the system will pinpoint the broken assumption just the same as in the case of the operation removal, and also explaining the specific reasons of the problem.

6. Conclusions

The reuse contracts model is a valid conceptual structure upon which reasonings can be made. These reasonings can be carried out in a static manner, without the need of testing or running (even building) the program under inspection.

The Itacio model was supposed to be flexible enough to fit different component notions and express very different kinds of restrictions. Applying the Itacio verification system to reuse contracts has been a demonstration of this fact, since Itacio had not been designed to cope with this scenario.

The combination of reuse contracts and the Itacio model can be of value (provided appropriate tools have been developed, which is a perfectly achievable goal as proven by the simplicity of the current prototypes) to verify the evolution of a system. It allows modelling its collaboration structure, keeping record of the assumptions made and automatically verifying that these assumptions and requirements are fulfilled.

7. References

[1] Cernuda, A., Labra, J. E., Cueva, J. M. *Itacio: A Component Model for Verifying Software at Construction Time*. III ICSE Workshop on CBSE. 5-6 June 2000, Limerick, Ireland. http://www.sei.cmu.edu/cbs/cbse2000/papers/ index.html

[2] Cernuda, A., Labra, J. E., Cueva, J. M. *A Model for Integrating Knowledge into Component-Based Software Development*. To appear in KM - SOCO 2001, 26-29 June 2001, Paisley (Glasgow, Scotland).

[3] Hall, Pat. *Educational Case Study – what is the model for an ideal component? Must it be an object?* III ICSE Workshop on CBSE. 22nd International Conference on Software Engineering. 5-6 June 2000, Limerick (Ireland).

[4] Holland, Ian. *The Design and Representation of Object-Oriented Components*. Doctoral Dissertation, College of Computer Science, Northeastern University, 1992.

[5] Itacio project web page. http://i.am/itacio

[6] Lucas, Carine. *Documenting Reuse and Evolution with Reuse Contracts*. Ph.D. Dissertation, Vrije Universiteit Brussel, Belgium. September 1997.

[7] Meyer, Bertrand. *Object-Oriented Software Construction (2nd edition)*. Prentice Hall, 1988.

[8] Prosise, Jeff. *Programming Windows 95 with MFC*. Microsoft Press.