Communicating Process Architectures 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

Formal Contracts: Enabling Component Composition

Marcel BOOSTEN
Philips Medical Systems, P.O. Box 10000, 5680 DA Best, Netherlands
Marcel.Boosten@philips.com

Abstract. Traditional component interaction is based on interface calls and callbacks. Such interaction can introduce integration faults, i.e., side effects at the moment of component integration. Solutions to such problems can be hard to apply, and may require drastic changes in the design of the involved components. This paper introduces Formal Contracts, a software construct that allows side-effect free component interaction, and thereby avoids the introduction of integration faults. Furthermore, via a state machine representing the inter-component contract, Formal Contracts, in addition to the static aspects, formally specify the dynamic aspects of component interaction. Formal Contracts are a pragmatic software mechanism that supports the full development cycle: from the specification and decomposition until the debugging, composition, and test of a system.

1 Problem Statement

Research, as in Trew[2], has identified that during software Component Integration, so typically at a relatively late stage of the development cycle, the following integration issues make the development cycle costly and unpredictable:

- Race conditions in component interaction
- Re-entrant callbacks
- State-inconsistency between components and unexpected state-event combinations

The integration issues listed are only revealed at the moment components are integrated into a larger whole. These problems are largely caused by the interface definitions between components being unclear and component interaction containing side-effects that only show up during component integration. This paper claims that by using a different mechanism for component interfaces, the Formal Contracts, all these problems can be avoided by design.

Furthermore, Formal Contracts facilitate pragmatic, but also formal, specification of component interaction, and, with negligible additional effort, facilitate automated testing. Formal Contracts therefore combine "design for testability"[1] with "design for ease of testing"[1], and support the whole of the development cycle.

Formal Contracts are based on the principles of CSP: using these principles, the integration issues are avoided by design, thereby enabling component composition. A Formal Contract can be considered to be a generalised form of a Channel, or collection of Channels. A Channel is an explicitly instantiated inter-component interface in Occam - a CSP-based programming language. The basic idea behind Formal Contracts is the addition of a formal - i.e., machine readable, executable, and verifiable - specification of the interaction between components to the explicitly instantiated interface between the components. Formal Contracts specify, implement, and test the dynamics of the interaction between components, and thereby pragmatically support the whole of the software development cycle.

2 Technical Aspects of Formal Contracts

In this section, we will introduce, illustrate, and motivate the technical aspects of Formal Contracts. This section is meant to give the reader an in-depth understanding of the combination of techniques and concepts used in Formal Contracts, such that at a later stage the advantages of the use of Formal Contracts can be motivated.

2.1 Contract, Role, Component

Formal Contracts define the interaction between a number of Roles. Each Role is an abstract entity that interacts with the other Roles that are involved in the Formal Contract. When instantiating a Formal Contract, each Role of the Formal Contract has to be fulfilled by an instantiated Component. A Role is an abstract base class or interface, i.e., a collection of methods.

2.2 Formal Contracts exist as components

A fundamental idea behind Formal Contracts is their explicit existence: each Formal Contract is what one would normally consider being a, even though relatively small, component in the system. I.e., a Formal Contract is a self-contained entity that provides a number of interfaces. The interfaces it provides are the Roles of the Formal Contract. The Components use the Roles, i.e. use the interfaces provided by Formal Contracts. This is illustrated with an example in Figure 1.

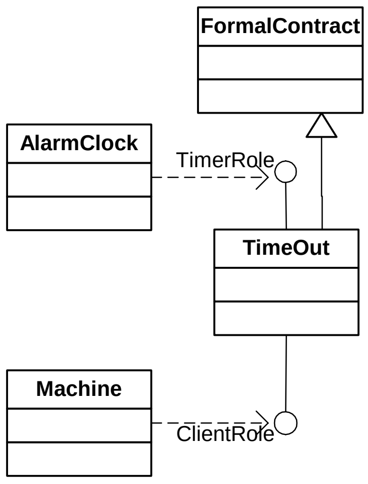

Figure 1: *The Formal Contract "TimeOut" with two Roles "ClientRole", "TimerRole", and two Components, "Machine" and "AlarmClock", that use, and thereby implement, their Role.*

Notice that this approach differs significantly from the standard approach in Object Orientation (OO) and component based infrastructures: Formal Contracts *are* the new interface between Components; Formal Contracts provide the interfaces for component interaction, whereas traditionally, Components would provide interfaces themselves, and directly use the interfaces provided by other components.

Traditional component interfaces are *asymmetrical*: a component is either a user or provider of an interface. Formal Contracts are *symmetrical*: Components have to fulfil the Formal Contract when using it.

Furthermore, notice that even though the approach conceptually differs significantly, still, standard component frameworks can be used to implement Formal Contracts.

2.3 Formal Contracts have explicit and observable state

Another fundamental idea to Formal Contracts is the fact that Formal Contracts have *state*, and that a Formal Contract *is a state machine* (or small program, if you like). Figure 2 illustrates interaction between two components in the traditional setting. Both components have an internal state, however, typically, components maintain an abstraction of each others state, the "externally observable state" to ensure that the messages send (or interface methods they call) occur as specified. This approach can easily become complicated, and is therefore error prone.

In Formal Contracts, we use a different approach: Formal Contracts have state. This state should not be considered to be the state of any of the components involved with the contract; no, it is the state of the Formal Contract itself. This way, implementers of Components need not implement abstractions of the state of other components; they can simply ask the Formal Contract for its current state, if they would want to know it. Formal Contracts export their state via read methods. These read methods -of course- do not change the Formal Contract's state.

The following code illustrates Formal Contract initialisation and the state read methods.

```
 class TimeOut : FormalContract
{
private:
  state : IDLE | COUNTINGDOWN | TIMEDOUT | CANCELLED;
public:
  TimeOut::TimeOut() { state = IDLE; }
  // Observable State
  Bool TimeOut::getState()
  {
    return state;
  }
  ...
}
```

2.4 Specifying and verifying dynamic behaviour

Another fundamental idea of Formal Contracts is that they specify the dynamic behaviour, simply by coding a small state machine that covers the full dynamic behaviour.

Formal Contracts define the relation between a number of Roles. Each Role consists of a number of state modification methods, that based on the parameters passed via the method, adjust the state of the Formal Contract. The state change is observable via the state read methods.

The implementation of the state modification methods within a given Formal Contract specifies formally the precondition and the effect on the state of the Formal Contract. Preconditions are verified via assertions, similar to Meyer's approach [8].

```
class TimeOut : FormalContract
        interface TimeOutClientRole
        interface TimeOutTimerRole
{
  ...
  int timeInSeconds;

  // Observable State
  ...
  int TimeOut::getTimeInSeconds() { return timeInSeconds; }

  // ClientRole
  void TimeOut::start(int aTimeInSeconds) {
    assert(state == IDLE);
    timeInSeconds = aTimeInSeconds;
    state = COUNTINGDOWN;
  }

  void TimeOut::cancel() {
    assert(state != IDLE);
    if (state == COUNTINGDOWN) {
      state = CANCELLED;
    }
  }

  // TimerRole
  void TimeOut::timeOut() {
    assert(state == COUNTINGDOWN);
    state = TIMEDOUT;
  }
}
```

The state diagram of this Formal Contract is shown in Figure 2.

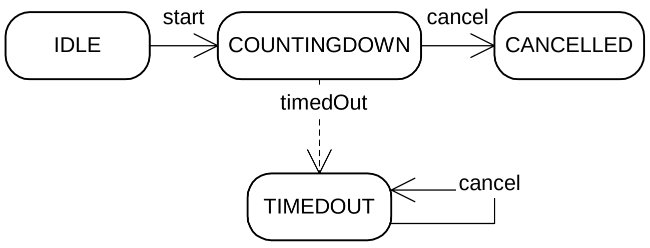

Figure 2: *State diagram of Formal Contract TimeOut. The lined arrows represent the transitions caused by the ClientRole, the dotted arrow represent the transitions caused by the TimerRole.*

In case the timeOut method is called while the state is IDLE, the assertion fails. The Component that plays the TimerRole can in that case easily be identified as the cause of the problem. Similarly, when the cancel method is called in state IDLE, the ClientRole is not well fulfilled. Typically, the Component that fails the Contract can easily be identified: either via the debugger, or via logging information of the method called and the current state of the Formal Contract. While integrating components, such assertions to pinpoint problems can help significantly.

Furthermore, while performing tests at component level, the Formal Contracts are to be present as well. Violations of Formal Contracts are to be eliminated during component-level tests.

2.5 Base Class for Formal Contracts extends component infrastructure

To make it even more explicit that Formal Contracts exist as implementation entities, a class is used as the base class for all Formal Contracts. Of course, functionality that is common by all Formal Contracts is implemented in that class. The Formal Contract class should be considered an extension of the component infrastructure.

2.6 Observer Pattern with Asynchronous Contents-Free Notification

Formal Contracts use the Observer pattern[9], but in a modified form. The Observer pattern is a very powerful decoupling and abstraction mechanism. It forms the basis for the popular Model-View-Controller pattern. The main disadvantage of the Observer pattern is the risk of introducing re-entrancy[3]. Re-entrancy problems are one of the major risks during integration testing[1]. Formal Contracts avoid the re-entrancy problem completely by design: Asynchronous Notification is used, instead of the Observer pattern's Synchronous Notification. Figure 3 illustrates Asynchronous Notification.

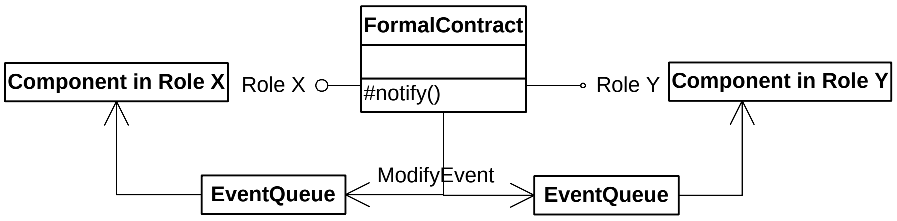

Figure 3: *Asynchronous Notification by communicating the event via a queue to the Observing Components.*

Another important aspect of the notification events used by Formal Contracts is them being contents free. The reception of a notification event indicates that the Formal Contract could have changed state (possibly it did not). It does not provide any additional information, for example not even an indication of the kind of state change that has occurred. By doing so, the Observing Components are forced to consult the Formal Contract to obtain the up-to-date status.

Due to the asynchronous (buffered) communication between Formal Contract and Observing Component, contents of a notification event would be likely not to resemble the state of the contract at that moment in time, misinterpretation of such contents could easily lead to state inconsistency problems and race conditions. These problems are avoided by using contents-free notification events.

Each Formal Contract notifies each of its state changes. This is illustrated in the following code example:

```
void TimeOut::start(int aTimeInSeconds)
{
  assert(state == IDLE);
  timeInSeconds = aTimeInSeconds;
  state = COUNTINGDOWN;
  notify();
}
```

The Formal Contract base class provides attach and detach functionality that allows Components to subscribe and unsubscribe to notification events generated by the Formal Contract. It is no problem for a Component to miss out on notification events that are generated before its first subscription to the Formal Contract: notification events are contents-free, by inspecting the state of the Formal Contract, newly subscribed Components can synchronise their internal state to that of the Formal Contract. It is the responsibility of each component to expect all specified series of state transitions (caused by other Components using the Formal Contract) to have occurred in-between inspections of the Formal Contract.

2.7 Consistency via a Reentrant Locking mechanism

In each Component infrastructure it is important to allow the use of multi threading and/or multi processing. As a consequence, Formal Contracts have to be instance MT-safe, i.e., safely usable by multiple threads or processes. Therefore, a locking mechanism is introduced: each method of each Formal Contract locks its Formal Contract during data access, and is therefore an atomic operation on the Formal Contract. Here are two examples:

```
void TimeOut::start(int aTimeInSeconds) {
  lock();
  assert(state == IDLE);
  timeInSeconds = aTimeInSeconds;
  state = COUNTINGDOWN;
  notify();
  unlock();
}

Bool TimeOut::getState() {
  enum theState;
  lock();
  theState = state;
  unlock();
  return theState;
}
```

However, to avoid race conditions, it can be important for a Component to base decisions on how to change the Formal Contract on the current state of that Formal

Contract. When the AlarmClock would use the following code, this would introduce a race condition:

```
if (contract->getState() == COUNTINGDOWN) {
   // Race condition here when contract would be cancelled,
   // since timeOut() is only allowed in the COUNTINGDOWN state.
   contract->timeOut();
}
```

The race condition occurs because the contract may change while decisions based on the contract's state are taken. We introduce a re-entrant locking mechanism to allow Components to avoid of such race conditions:

```
contract->lock();
  if (contract->getState() == COUNTINGDOWN) {
     contract->timeOut();
  }
  contract->unlock();
```

The locking mechanism is re-entrant; this means that lock/unlock pairs on the same Formal Contract can be nested without causing deadlock. Often, the use of the locking mechanism can be avoided by choosing the state modification methods --and therefore the state machine itself-- carefully. For example, by changing the `timeOut()` method from:

```
void TimeOut::timeOut() {
  lock();
  assert(state == COUNTINGDOWN);
  state = TIMEDOUT;
  unlock();
}
```

into:

```
void TimeOut::timeOut() {
  lock();
  if (state == COUNTINGDOWN) {
    state = TIMEDOUT;
  }
  unlock();
}
```

Then, to avoid the race condition in the AlarmClock, no locks are needed anymore. By introducing the design rule that a Component are allowed to have locked at most one Formal Contract at any time, deadlock problems are avoided.

Note that the example in this section is simplistic. While developing a Formal Contract, attention should be paid to make the Roles as intuitive as possible, and to avoid the need for lock/unlock outside the Formal Contract, so in the calling Component, as much as possible. From a testability point of view, one can also argue that Formal Contract should contain as little assertions as possible, and that therefore changes that replace an assertion by an if statement are always (or typically) to be preferred.

The sequence diagram in Figure 4 illustrates the interaction with the Formal Contract. The diagram shows how the Machine starts the AlarmClock via the formal contract by (1) changing the state of the contract, (2) the AlarmClock being notified of a changed contract, (3) the AlarmClock discovering that via *getState* that it has to start its internal timer. Furthermore, the diagram shows a situation in which the *cancel* of the timeout and the *timedOut* occur simultaneously: the Formal Contract decides which of the two events took place first (*cancel* in this example), and notifies both the Machine and the AlarmClock of this fact. This way, all possible inconsistencies due to differences in interpretations from either side are avoided: both Machine and AlarmClock have the same view on the situation that occurred; they both find out that the TimeOut has been cancelled.

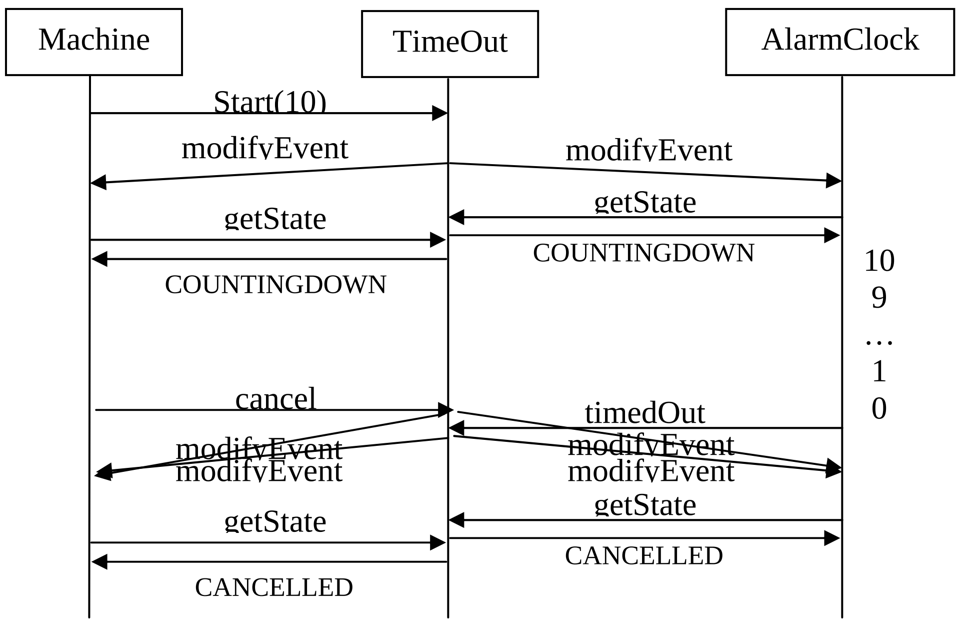

Figure 4: *Sequence diagram illustrating interaction between Machine, the Formal Contract TimeOut, and the AlarmClock.*

2.8 Shared ownership via reference counting

Formal Contracts are shared by the Components fulfilling the Roles. Life cycle management of Formal Contracts is organised via an access/de-access mechanism based on reference counting. The Component that is last to de-access the Formal Contract, automatically de-allocates it. Other solutions, for example based on garbage collection, are also fine solutions.

3 Advantages of Formal Contracts

3.1 Avoidance of State Inconsistencies and of Race Conditions

Each Formal Contract is always in a specified state, i.e., in a state that can be reached by the state machine that implements the Formal Contract. All inter component communication takes place via the Formal Contracts and their contents-less notification mechanism. Components can read and change the Formal Contract's state and thereby communicate: inconsistencies and race conditions can easily be completely avoided, while facilitating the use of multi threading and other forms of concurrency.

3.2 Avoidance of Re-entrancy Problems

Threads of different Components will *never* `pass through' a Formal Contract, and therefore never enter the execution domain of other Components. Each Formal Contract acts as a thread barrier. It allows communication and synchronisation between threads, however, it stops threads from entering each others domain. This way, inconsistency and re-entrancy problems related to multi-threading across component boundaries are avoided by design.

Often, in sequential OO programming, the use of anonymous call-backs are the cause of re-entrancy problems. However, even in situations where direct interface calls are used, re-entrancy problems[5] are likely to occur. Formal Contracts eliminate these problems by design, and are therefore also important for usage in single threaded applications.

3.3 Formal Contracts during the development lifecycle

Integration testing often has to take place relatively late in the project life cycle. It is crucial to avoid integration and integration test problems by design, since at the stage in the project that integration problems are discovered, it is typically too late to solve the problems without project slippage. Improving in the area of system integration is important for many organisations developing large and complex systems. This is only possible via an integral approach that addresses the full development cycle, and not just the last phase - at which time it is too late to solve the problems structurally:

- **Interface Specification and System Decomposition** Formal Contracts, or actually the state machine inside, is the formal specification of the interaction between Components. In addition to the definition of the static aspects of component interaction via method names and corresponding parameters, Formal Contracts also specify fully and formally the dynamic aspects of the Component interaction.
- **Component Development and Component Testing** Formal Contracts facilitate automated testing right from the start of the component development until the end of the integration phase. In principle, the "output" of each Component is automatically and always tested by the Formal Contract. Furthermore, by observing the state of the Formal Contracts that are to be fulfilled by a specific Component, a lot of insight in the behaviour of the Component becomes visible. Formal Contracts therefore are a powerful debugging tool as well. Furthermore, Formal Contracts can facilitate deadlock analysis across multiple Components. The state of each Formal Contract provides insight in "who is waiting for who". Furthermore, I believe that techniques for deadlock cycle detection in designs can exploit Formal Contracts, and thereby make automated deadlock detection possible even for large real-life component structures. Of course, significant research would be needed to proof this, and to fill in the many more details.

The use of Formal Contracts requires a new way of thinking about Component interaction. At first sight, it might look like a lot of administrative overhead. And, indeed, for very small and trivial interfaces it does introduce significant administrative overhead. However, Formal Contracts are meant as specification, communication, and verification medium between different Components, so between groups of people.

At that point, Formal Contracts become a very neat, powerful, and low-overhead tool. My own experience has shown that the approach is also applicable at a much smaller scale, so as interface between "one" or "few man" components.

4 Comparison to State-of-the-Art Techniques

4.1 Interface Definition Language (IDL)

Today, the IDL is widely used to specify component interfaces. IDL is limited to the definition of groups of methods or functions. Formal Contracts extend IDL with a means to formally specify the dynamic interface behaviour, and to verifying the Component's compliance to it. Furthermore, in contrast to IDL, Formal Contracts has been designed to avoid race conditions, re-entrance problems, and state-inconsistencies.

IDL is a language independent of any specific programming languages. It provides language de-coupling between components. This aspect of IDL can be used for Formal Contracts as well: Formal Contracts are (small) components of which the interfaces methods, the Roles, can be published in IDL or some other interface definition language used by the component infrastructure.

4.2 Model - Observer based Decoupling

Model - Observer based decoupling is sometimes used for the definition of component interfaces. Compared to Formal Contracts, these approaches typically do not eliminate race conditions and re-entrancy problems via the design of the Component interconnect. Furthermore, these approaches typically do not model the dynamic interaction at all. Typically, the Model that contains the state shared between Components is a data structure of which the fields can be changed any time into any of its values. Components fulfilling the interface should be able to deal with such state changes. The dynamic aspects of the interface are not well covered by such approach. Typically, the approach allows all state transitions on the model, and therefore the dynamic behaviour need not be specified – an approach which is practically unusable in case of non-trivial control applications.

4.3 Trew's Design for Testability approaches

In his paper[4], Trew describes design patterns that can be applied to eliminate the typical integration problems: race conditions, re-entrancy problems, state inconsistencies. Formal Contracts are a very specific combination of such design patterns. By using Formal Contracts for all interactions between Components, many of the difficult integration problems can be avoided. However, note that, even if Formal Contracts would be applied rigorously in a design, several of the design patterns identified by Trew would still be needed for more specific or exceptional situations.

Formal Contracts address the main integration problems identified by Trew, however, they also introduce a formal way of component interaction specification that decreases the chance of interpretation problems. Furthermore, Formal Contracts can very well support a new way of working: they can become the leading specifications during decomposition, and the glue during composition. Formal Contracts integrate system decomposition, interaction specification, interaction verification, and component composition into a streamlined approach.

4.4 Communicating Sequential Processes (CSP)

Within CSP research groups [5][6], there is a very clear awareness that component interaction should takes place across well-defined side-effect-free interfaces. Furthermore, they noticed that today’s Object Oriented and Component infrastructures do not provide such well-defined interfaces. Re-entrancy and concurrency problems violate the promised and promoted "Encapsulation" characteristic that should enable black box reusable components and facilitate assembly of systems by component composition.

The work presented in this paper is based on the fundamental ideas behind CSP, and has been inspired by the work of the research group to improve interaction definitions, but, at the same time, takes industrialist's experience and way-of-working into account.

Comparing the work presented in this paper with work performed on CSP is difficult due to the variety of work performed. It is important to notice that Formal Contracts use the CSP principles, therefore, it is possible to use the mathematics of CSP to specify and analyse properties of systems constructed with Formal Contracts as well.

If we compare the work performed here with work performed within the CSP-based research groups that focus on an embedding of CSP principles in programming languages, we clearly see a number of differences.

- **XToYChannels** Within JCSP and JavaPP, a relatively large number of Channels is introduced: OneToOneChannel, OneToManyChannel, ..., AnyToAnyBufferedChannel. The approach presented here generalises this to a user-definable state machine. The advantage of the state machine is that it combines logically dependent Channels into a single whole of which the dynamic behaviour is both formally and understandably specified.
- **Channels are typically only half a Contract** Channels are communication primitives between Processes. However, since channels are typically unidirectionally in Occam, JCSP and JavaPP, in order to specify the dynamic behaviour between two Processes, typically one would have to define a set of sequences of interaction across a *number of* channels between the two processes. A Contract is the complete relation between two processes, and is N-directional.
- **Event based vs. Process Based** The approach presented here allows both Process-based implementations and Event-based implementations: an Event-Based process is a main loop with a large ALT inside in which all events are handled. The decision between an Event based vs. Process based should be made on a per case bases.
- **Formal Contracts enabling combined input output ALTs** In contrast to all the CSP-based implementations that are currently available, Formal Contracts enable both input and output ALTs, and, just as natural, combinations of the two. A combined input output ALT could be the following:

```
do {
  WaitForEvent(Contract1, Contract2);
  keepwaiting = FALSE;
  if (contract1->canWrite()) {
    contract1->output(value1)
  } else if (contract1->canRead()) {
    value2 = contract2->input()
  } else keepWaiting = TRUE;
} while (keepWaiting);
```

- **Formal Contracts combine synchronous and asynchronous communication** The CSP-based approaches are mostly based on synchronous communication. Formal Contracts combines the two: communication with the contract is synchronous but basically non-blocking. The communication of the modification events is asynchronous, and 'overwriting'. Processes can decide to perform a blocking wait on modification events. The author of this article is convinced that the model presented here can easily be modelled in CSP, especially because the asynchronous communication of modification events is 'overwriting'. In Formal Contracts, Processes typically do not synchronise with each other, only with the Formal Contract. Compared most existing CSP-based approaches, this has the advantage that by default synchronisation (which can be costly, since it reduces parallelism) is avoided.

4.5 Jonkers' API Structure and Model-Based Specification

In his paper [7], Jonkers presents the API Structure and Specification method used for DVP2. There are several similarities between his approach, and the approach presented in this paper: he recognises and identifies contracts as combinations of different interfaces; he also identifies roles, and makes components fulfil those roles. He also recognises that the contract is an excellent way for people to agree on the interface.

Jonkers uses a Model-Based approach to specify the behaviour of each role: "abstract implementations" of each Role are used for specification purposes. Formal Contracts completely inverse this specification: instead of providing "abstract implementations" for each Role, the true implementation of the interaction is specified by implementing a Formal Contract. This way, the obligations and expectations of each Role and their interdependencies are made explicit. Formal Contracts are integrated as small components into the system: they are more than specification, they are the interaction medium. Formal Contracts facilitate the verification of the behaviour of each Component: the Formal Contract checks input events versus its internal state. Using Jonkers' approach, verifying that a Component fulfils its Model, its "abstract implementation", is much more complicated. The Formal Contracts approach can very well be combined with a Model-Based development approach, in that case, it can be considered an extension to the Model-Based approach. Formal Contracts avoid race conditions, re-entrancy problems, and state inconsistencies. These fundamental integration aspects have not been addressed by Jonkers' approach.

In other work, Jonkers has used Interaction Objects for communication and synchronisation. The objective of this research was different: it was meant to minimise the coupling between Processes aiming at minimisation of communication latency and throughput impact on the whole of the system.

5 Background, Context, and Experience so far

The solution presented in this paper is the result of accumulated fragments of experience in the area of component specification and integration. The first inspiration for this approach has come from the 1998-2000 Communicating Process Architectures (CPA) conferences, especially due to presentations and discussions with Peter Welch, Andrew Laurence, and Tom Locke. Thanks, it has always been very interesting!

During my working period at Philips Medical Systems, Product Management Group (PMG) Computed Topography (CT), I was faced with component integration problems, and with multi-threading aspects. I would like to thank my colleague CT architects at that time: Peter Jaspers, Phil van Liere, Jacco Wesselius, Joland Rutgers, for being as stubborn as myself, and thereby -in the end- pushing me to find the solution as described in this paper. I have implemented the approach described in this paper in the multi-threaded so-called "Multi-Scan" controller in the CT software; one of its most complex parts. The implementation demonstrated a significant reduction in complexity compared to an approach based on asynchronous message passing, in which the multitude in race conditions was hard to handle.

Furthermore, I'm very grateful for Tim Trew's work. He has very well used his architectural experience to identify the major integration problems in practical industrial situations. In my experience, it was extremely difficult to convince highly experience architects of the problems at hand, and of the need for a structural solution. However, if the consequences are in the order of many man-years of additional effort, convincing people will be easier. Tim Trew's award winning “most influential” paper[1] convinced people of the relevance of such work. Consequently, I decided to write this paper on Formal Contracts down. Some results of discussions with Tim Trew and a quick discussion with Hans Jonkers have been incorporated in this paper.

6 Conclusions

Formal Contracts are a software construct enabling side-effect-free component interaction. Thereby, Formal Contracts avoid the introduction of major integration faults. Furthermore, Formal Contracts specify in a human readable and machine executable form the dynamic aspects of component interaction. They check the state of the contract versus incoming modification requests. These characteristics make Formal Contracts a pragmatic tool that supports the full development lifecycle: from the specification and decomposition until the debugging, integration, and test of the system.

7 References

[1] Tim Trew. Demystifying Design for Testability. Philips Software Conference June 2002.
[2] Tim Trew. The aims of Integration Testing. PRL Tech. Note 3922. November 1999.
[3] Tim Trew. Software Component Composition: Still 'Plug & Pray'? Philips Software Conference 2001.
[4] Tim Trew. Testability of Component Based Software: Mastering Component Interaction. PRL Int. Note
[5] Tom Locke. The Broken Promises of OO, And How We Might Do Better. Slides presented during one of the SIG evenings at the CPA 2000 conference.
[6] Peter Welch. Java Threads in the light of occam/CSP. WoTUG-21. IOS Press, 2000. ISBN 90-5199-391-9
[7] Hans Jonkers. DVP2-API: Structure and Specification. Philips Software Conference June 2002.
[8] Bertrand Meyer. Object Oriented Software Construction. Prentice Hall, 1997. ISBN 0-13-629155-4
[9] E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.