An alternative to Quiescence: Tranquility

Yves Vandewoude*, Yolande Berbers
Department of Computer Science
KULeuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
{yvesv, yolande}@cs.kuleuven.ac.be

Peter Ebraert*, Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium
{pebraert, tjdhondt}@vub.ac.be

Abstract

This paper revisits a problem that was identified by Kramer and Magee: placing a system in a consistent state before and after runtime changes [16]. We show that their notion of quiescence as a necessary and sufficient condition for safe runtime changes is too strict and violates the black-box design principle. We introduce a weaker condition, tranquility; easier to obtain, less disruptive for the system and still sufficient to ensure application consistency. We also present an implementation of this concept in a component middleware platform.

1 Introduction

An intrinsic property of a successful software application is its need to evolve. In order to keep an existing application up to date, we continuously need to adapt it. Usually, evolving such an application requires it to be shut down, because updating it at runtime is generally not possible. In some cases, this is not acceptable. The unavailability of critical systems, such as web services, telecommunication switches, banking systems, etc. could have unacceptable consequences for the companies and their position in the market.

A solution for this problem lies in the dynamic adaptation of the software system, in which a part of the system is updated while it is active. To do so, however, the system must reside in a consistent state before the change is performed [16]. A consistent state is a state from which the system will be able to terminate correctly.

A system is viewed as moving from one consistent state to the next, as the processing of transactions continues. During their execution, however, the state of these transactions is distributed in the system and may temporarily leave different nodes in a mutually inconsistent state. Therefore, nodes can not always be updated during the execution of a transaction without breaking application consistency.

*Authors funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

2 The concept of quiescence

The topic of when a piece of software is in the appropriate status for undergoing an update (further referred to as updatability) has been the focus of much research in the past. Very influential in this regard was the work of Kramer and Magee [16]. In their model, a system is seen as a directed graph whose *nodes* are system entities and whose *arcs* are connections between those entities. Nodes can only affect each others states via *transactions*, which consist of a sequence of messages that must be executed atomically (i.e. either all messages are executed, or none of them are). The node that starts the transaction is referred to as the *initiator* of the transaction. The model of [16] assumes that transactions complete in bounded time and that the initiator of a transaction is aware of its completion.

Kramer and Magee abstract the status[1] of an application into a set of different configuration statuses for each node and consider two main statuses for each node: active and passive, whose definitions are as follows:

Definition 1 (Active Status) *A node in the active status can initiate, accept and service transactions.*

Definition 2 (Passive Status) *A node in the passive status must continue to accept and service transactions, but:*

(i) it is not currently engaged in a transaction that it initiated

(ii) it will not initiate new transactions

[1]Kramer and Magee used the term *state* instead. In this paper, we choose to distinguish between the internal *state* of a node, and the *status* which describes its condition in relation to the evolution process.

They identify a passive status as a necessary, but insufficient condition for updatability, as a node may still be processing transactions that were initiated by other nodes. Therefore, they introduce a stronger concept:

Definition 3 (Quiescence) *A node is a quiescent status if:*

(i) it is not currently engaged in a transaction that it initiated

(ii) it will not initiate new transactions

(iii) it is not currently engaged in servicing a transaction

(iv) no transactions have been or will be initiated by other nodes which require service from this node

Although quiescence is a sufficient condition for updatability, it has the problem that enforcing quiescence causes serious disruption to a system for even the smallest update. Not only must the node which is to be updated be put in a passive status, but this is also the case for every node that is directly or indirectly capable of initiating transactions on this node. This brings along a serious drawback with respect to change impact [4].

Furthermore, the approach by Kramer and Magee assumes that all parties involved in the transaction are aware of their involvement in a transaction. This implies that each node is not only aware of the services that it provides, but also about the party that requests this service. This clearly increases the coupling between the involved parties, and hinders reusability [19].

It are those problems that we address by introducing the concept of tranquility.

3 The concept of tranquility

A good approach for enhancing reusability and decoupling the system parts, lies in a black-box design of system nodes. This implies that nodes may require services from other nodes they are connected to, but may never rely upon their implementation. If all nodes are black-box by design, all participants of a transaction are either the initiator of the transaction or directly connected (adjacent) with the initiator. Nodes that are indirectly connected with the initiator can by definition not participate in a transaction driven by the initiator, since their existence is unknown to the initiator. Note that any participant of the transaction can in turn initiate new transactions in response to a message they process. These *sub-transactions*, however, are not known to the original initiator.

This black-box property is exploited by the concept of *tranquility*, which we introduce as an appropriate status for updatability:

Definition 4 (Tranquility) *A node is in a tranquil status if:*

(i) it is not currently engaged in a transaction that it initiated

(ii) it will not initiate new transactions

(iii) it is not actively processing a request

(iv) none of its adjacent nodes are engaged in a transaction in which this node has already participated and might still participate in the future

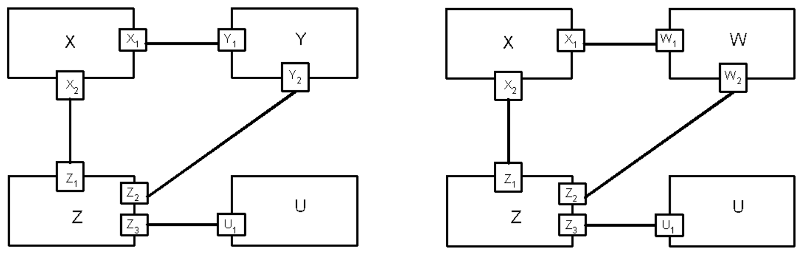
Quiescence is a stronger concept than tranquility in the sense that quiescence implies tranquility but not vice versa. Condition (iii) of quiescence implies that the node is neither actively processing a request nor waiting for a new request in an already active transaction. This trivially implies the condition (iii) of tranquility. Condition (iv) of quiescence states that none of the adjacent nodes have initiated or will initiate a transaction in which N participates. Hence, no such transaction is active, trivially implying condition (iv) of tranquility. Tranquility does not imply quiescence however, since it does not require that nodes connected with N may not initiate new transactions that involve N. For tranquility, nodes directly connected to N must not achieve a passive status.

The tranquility condition has the distinct advantage that nodes can participate in transactions without being aware that their actions are part of this transaction. In addition, tranquility is much less disruptive than quiescence since only the affected node N must be passivated. Although the third condition of tranquility requires some adjacent nodes to finish a certain transaction, these nodes must not be completely passivated for the tranquility condition to uphold.

4 Tranquility as a sufficient condition for updatability

Although a weaker condition than quiescence, tranquility is nevertheless a sufficient condition for updatability when two basic assumptions associated with an application are valid. Firstly, we assume that both the original and the resulting configuration of nodes are correct (incorporating a consistent application state). Secondly, since each node should be reusable, it should only rely on external functionality if this functionality is declared to be public. This can only be achieved if the communications between nodes is made explicit.

A node in the tranquil status is by definition not executing code. In addition, although a transaction in which it is involved may still be ongoing, its participation is either (1) finished, (2) not yet begun, or (3) part of a sub-transaction. In the first case, the update is clearly valid. In



(a) Original Configuration (b) Resulting Configuration

Figure 1. An example component configuration before and after an update.

the two latter cases, the validity of the update follows from the assumptions of a valid resulting configuration: transactions that have not yet begun may be executed by the new version.

For clarity, we illustrate this principle with two examples:

4.1 First example

Consider the component-oriented system from Figure 1. Assume that one wishes to replace component Y with a new version W. Furthermore, in the compositions shown in Figure 1, component X can execute a task for which it requires the assistance of its adjacent components. The transaction that realizes the execution of this task is shown in Figure 2. In Figure 2(a), the transaction is shown as it is executed in the current component configuration, whereas Figure 2(b) shows the same transaction from the point of view of X's implementation. Note that the subtransaction initiated by Z is unknown by X.

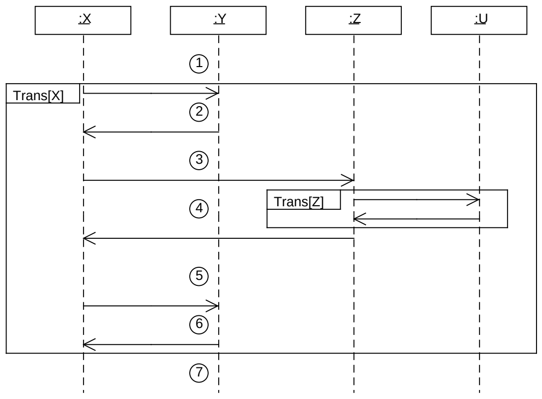
Assuming there are no other transactions specified by either X or Z (Y's adjacent components), the only times that Y can be safely replaced are identified on Figure 2 by the numbers 1 and 7. The periods identified by numbers 2 and 6 are characterized by execution in Y itself, and are therefore not suitable for replacement. Periods 3, 4 and 5 are entirely equivalent from Y's perspective: Y is an inactive but unknowing participant of a transaction initiated by X.

The replacement of Y will not change the transaction itself, since the transaction is entirely specified in X. Based on the validity of both the original and the resulting component configuration, this transaction will lead to a valid result with either Y or W (Y's replacement) as the adjacent component connected to X. However, a valid result is not guaranteed if the transaction starts with the old version of Y and finishes with the new version. This inconsistency occurs when Y supports two symmetrical operations, which are orthogonal to the working of X, but are nevertheless interrelated. For example, suppose Y is a (de)compression component, which offers two methods: `compress` and `decompress` that return a (de)compressed version of the data supplied by the sender of the message. Component X may wish to compress input data at the beginning of its task and decompress it again when it is done. Although it does not matter which compression algorithm is used (indeed, the transaction is valid with both the old and new version), correct functionality is not guaranteed if Y is replaced by another component in the middle of the transaction. Note that this is the case whether or not Y is a stateless or stateful component.

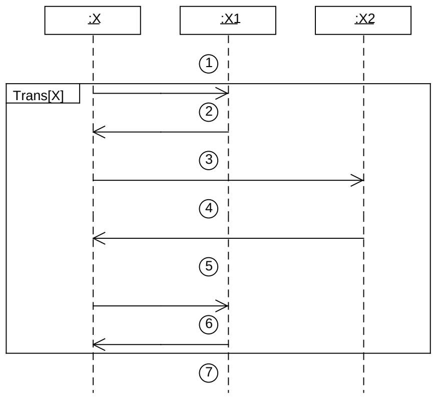
In this transaction, both the condition of quiescence and tranquility forbid replacement at the times 2-6. However, the tranquility condition allows replacement of Y at the beginning or the end of the transaction (times 1 and 7). This is not the case for quiescence, as quiescence also requires that *no transactions have been or will be initiated by other nodes which require service from this node*, which requires X (and Z for that matter) to be passivated completely. Exploiting the validity of the resulting composition, the tranquility requirement allows much quicker replacement while still ensuring consistency.

4.2 Second example

A slightly more complex scenario is shown in Figure 3. This scenario assumes the same initial and resulting component configuration, but assumes a different active transaction. Looking at Figure 3(a), one might be inclined to think that there is a fundamental difference with the previous example. At time 4, Y may be replaced according to our tranquility definition, since Y is currently not involved in a transaction that it initiated, the transaction by Z has not started yet, and further execution of the transaction by X no longer directly involves Y from X's point of view. As it turns out, it is indeed correct that Y can be replaced at this point. This is shown in Figure 3(b). The transaction initiated by Z is independent from the transaction initiated by X. Since both the initial and resulting configuration are correct, the transaction of Z leads to correct results using either Y or W as its participant. The ongoing transaction initiated

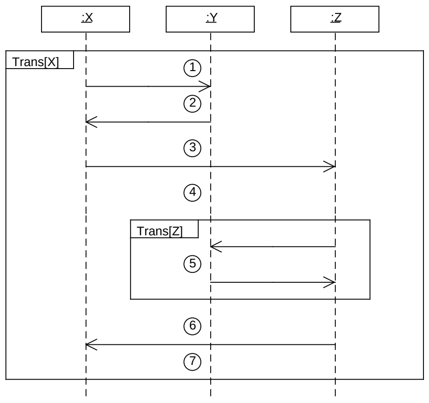


(a) Transaction as Executed

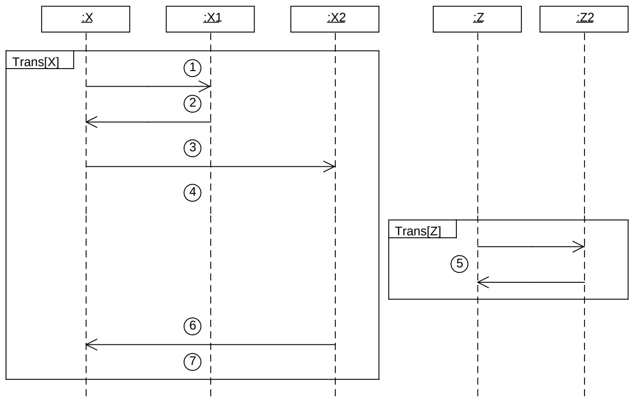


(b) Transaction as perceived by X

Figure 2. A transaction in which Y participates, prevents Y from being updated



(a) Transaction as Executed



(b) Transaction as perceived by X and Z

Figure 3. Y is used in a transactions from X and in a subtransaction initiated by Z

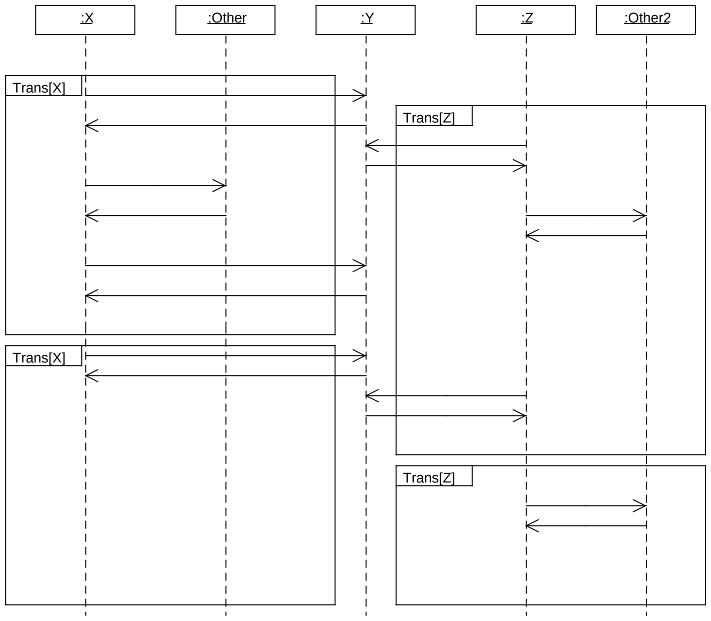


Figure 4. A scenario in which component Y will never reach tranquility.

by X is unaware of the transaction initiated by Z. Due to their independence, it is perfectly possible that the transaction shown in Figure 3 starts with Y and finishes with the new version W. A replacement of Y at time 4 is not permitted using quiescence as a condition, since quiescence does not take into account the independence of both transactions.

5 Reachability of the tranquility condition

Although tranquility is a sufficient condition in order to guarantee application consistency during an update, there is one important disadvantage: it is not guaranteed that the component which is to be updated will ever reach this status. This is the case when the component is used in an infinite sequence of interleaving transactions. An example of such case is shown in Figure 4. The Figure shows two interleaving transactions that are infinitely repeated (only the first and the beginning of the second iteration are shown). Because Y is always active in a transaction in which it still needs to participate, it can never reach tranquility without directing X and Z to a passive status (which would imply quiescence).

Therefore, any system that implements dynamic updates using the tranquility condition must implement a fallback mechanism for when this condition is never reached. It should be noted that these situations are rather rare in practice ([10] pages 428-429) and that in most cases the tranquility condition occurs within a short period of time.

6 Implementation on component middleware

A prototype implementation was developed as an extension to a general purpose component middleware platform: DRACO. The implementation allows the middleware to drive active components to a tranquil status upon demand. When this status can not be reached, it transparently falls back to the quiescence requirement. We begin this section with the introduction of the main concepts supported by the DRACO methodology. A full description of either the component model, the language or its toolchain, is outside the scope of this paper and we restrict ourselves to the core concepts of the methodology and how these concepts map to the model by Kramer and Magee. Relevant implementation aspects of the component middleware environment are discussed in section 6.2. Finally, we present a detailed description of how the Live Update Extension Module (LUM) realizes updatability using the tranquility condition.

6.1 The DRACO component methodology

In DRACO , *components* are units of functionality which are implemented as a highly cohesive group of Java classes. Once instantiated, they represent a tightly coupled group of objects. Interconnection between components is achieved by means of *connectors*. According to [1], a connector is a reusable design element that supports a particular style of component interactions. DRACO assumes the interaction style that was defined in the SEESCOA project [2, 20]. In this model, components communicate by asynchronously sending messages through external interfaces that are formally specified using *ports*. Connectors attach to these ports and implement a pipe-like construct, which makes relaying or intercepting communication easy to achieve. The conditions of explicit communication that were assumed in section 4 are therefore clearly met in the DRACO component model.

In order to map our component model onto the model assumed by Kramer and Magee, it suffices to consider our components to be the nodes and our connectors to be the arcs of their directed graph. The bidirectional nature of connectors can easily be modeled using two directed arcs with opposite direction. Furthermore, in the DRACO component model, the state of components can only be changed by message interaction with other components, and all message sequences complete in bounded time.

6.2 An extensible middleware platform

The DRACO middleware platform was designed with extensibility in mind and offers an extensive API which can be used by extension modules to change the behavior of the core system. Its architecture consists of 5 core modules: (i) the component manager, responsible for loading and instantiating component instances; (ii) the message manager, responsible for the message delivery process; (iii) the scheduler, responsible for scheduling messages that have been sent and that are awaiting execution; (iv) the connector manager, responsible for (dis)connecting ports; and (v) the module manager, responsible for adding extension modules to the core system at runtime.

Message delivery is achieved in three stages. The first stage is the transmission of the message by the originating component. In this stage, the message passes through a sequence of message handlers (who can transform messages that pass through it) until it is handed over to the scheduler. This first stage is executed by the thread currently active in the originating component. Because interaction with the scheduler is non-blocking, message sending is asynchronous. In the second stage, the message awaits its execution inside a messagequeue from the scheduler. Finally, in the third stage, the scheduler's *worker-threads* continuously fetch messages, pass these messages through a similar handler chain and finally deliver them to the receiving component. The scheduler guarantees that the order of messages over a given connector is preserved and that messages are delivered sequentially.

The message handler mechanism opens up the delivery process as extension modules can insert or remove custom handlers that change the default behavior. DRACO also makes extensive use of the observer pattern, and allows for extension modules to subscribe themselves to a large number of events that are triggered before and after all important actions, such as component (un)loading, (dis)connecting and message sending.

6.3 Live update extension module

The Live Update Extension module (LUM) is an extension to the core DRACO system that allows components to be replaced by a new version at runtime. After the application maintainer has specified that a certain component C needs to be replaced, the LUM places that component in a tranquil status. The module then transfers the state from the old to the new version, rewires the connectors and activates the new version. This paper is only concerned with the first step and the following sections describe how the tranquil status is reached, and how the module falls back to quiescence if tranquility is not attainable.

6.3.1 Enforcing passivity

Since tranquility encompasses all requirements of passivity, the LUM will first direct C to a passive status before it enforces the other tranquility conditions. This passive status is attained by ensuring that:

(i) The component is not actively executing a message

(ii) The inflow of new messages to the component is restricted

If no messages are executing, the first passivity requirement is trivially fulfilled. In addition, no new transactions can be initiated by C, because messages can only be sent out by a component as part of code execution which in itself is triggered by a message.

The LUM achieves passivity by restricting all incoming traffic to C. It does so by replacing the standard delivery message handler on the receiving message chain of each port of C by a custom delivery message handler (see Figure 5). Although Figure 5 only shows one connected port on C, the situation is analogous for all other connected ports. To guarantee that all interactions with C are controlled, the LUM registers itself with the connector manager to temporarily prevent changes to the connections of C's ports. The LUM then sends a `Freeze` message to a random connected port of C.

As illustrated in Figure 5, any number of messages with component C as their destination can be present in the scheduler queue at the time the `Freeze` message is sent out by the LUM. Since the custom delivery message handlers are only inserted in the chains associated with ports of component C, messages that are intended for other components are unaffected by the replacements in the delivery message chains, which reduces unnecessary overhead.

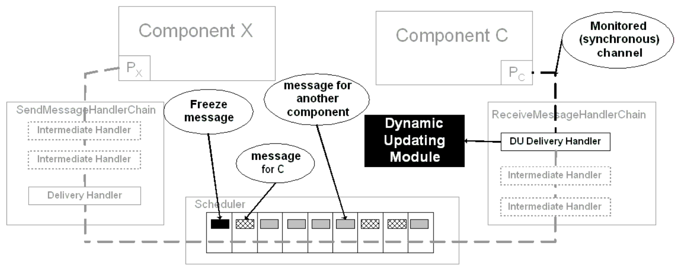


Figure 5. Situation after initiation of the update of C.

The custom handler introduced by the LUM initially mimics the behavior of the original delivery handler: it executes the method associated with the message on the component and then terminates (thus returning control through the delivery chain all the way back to the scheduler). This behavior changes after it encounters the `Freeze` message. At that time, the message is executed if it is supported by the component (allowing the designer of the component to specify custom cleanup code) or ignored otherwise. Afterwards, however, control is no longer returned to the scheduler. In addition, other custom delivery handlers associated with a port of C halt before executing the message, effectively terminating all communication with C.

Although no new messages can reach C, the passivity requirements have not been fulfilled so far:

First, the component may still be executing code in a dedicated thread. For brevity, we did not include a full description of how such threads are handled by the component system. For this paper, it suffices to say that the scheduler of DRACO is aware of such threads and that it can safely preempt the majority of such threads without leaving the component in an inconsistent state. Whether or not DRACO was able to preempt the thread, the LUM delays the update of the component until its execution has terminated. We refer the reader to [21] for more details.

Second, a transaction initiated by C can still be active. For example, at time 3 on Figure 2(a), component X is inactive and awaiting a response from Z. This leads us to the problem that the LUM must be able to determine when a component is actively engaged in a transaction it initiated. When a transaction consists of asynchronous messages (which is the case in DRACO), this can not be determined automatically unless the component that drives the transaction provides this information. Our implementation assumes that each component implements a method that returns whether or not the component is active in a transaction it initiated. The LUM queries this information and resumes message delivery on a message-per-message basis as long as C is in such a transaction. After each message, the expression embedded in C is reevaluated until the component has terminated its transaction. At this moment, C has achieved passivity, and will maintain this status since no further messages are allowed into C.

6.3.2 Ensuring the additional tranquility constraints

Before the passivated component C may be replaced, the additional conditions of tranquility must be met. The LUM does this by querying all adjacent components of C and determining whether these components are involved in a transaction they initiated. If so, the LUM requests from each of these components a list of all ports that have participated in their transaction, and a list of those ports that are still required to finish the transaction[2]. If a port of C is attached to a port present in both lists, the requirements of tranquility are not fulfilled.

The LUM then starts to monitor all messages entering the adjacent components of C by replacing their message delivery handlers. Whenever a message is delivered to these components, the transaction requirement is reevaluated. As the completion of the transaction requires the participation of C, it is necessary that C accepts messages involving the transaction. The custom delivery message handlers associated with ports of C will therefore resume message delivery, again on a message-per-message basis. After each message delivery, both to C and its adjacent components, the conditions are rechecked. In the majority of cases, complex systems of interleaved transactions or transactions with circular dependencies are not an issue ([10] pages 428-429) and a tranquil point is reached for C in a relatively short period of time. Once this tranquil point is reached, all messages to C are prevented and the tranquility condition is preserved for the duration of the update.

6.3.3 Fallback to quiescence

Because reaching tranquility in bounded time can not be ensured in general (section 5), the LUM also keeps an internal timer. If after a predetermined timeframe, tranquility was not reached for C, the system falls back to the more stringent and invasive requirement of quiescence which was proved to be reachable by Kramer and Magee in [16]. Their model assumes that a node has knowledge of whether its actions are part of a transaction initiated by another node. As this assumption is not valid in the our component model, ensuring the reachability of quiescence in bounded time needs to be examined further.

The problem is caused by dependent transactions, which Kramer and Magee define as follows:

[2]While it may seem a large overhead to generate this data, the information can be automatically generated from a state machine that describes the transaction. This state machine can be automatically derived from message sequence charts such as those used in the figures of this paper.

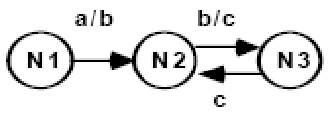


Figure 6. A system with cyclic dependencies

Definition 5 (Dependent Transaction) *A dependent transaction is a two-party transaction whose completion may depend on the completion of other consequent transactions.*

In other words, t_i is a dependent transaction if there exists a chain of transactions $t_i, t_j, \ldots, t_s$ in which each, with the exception of t_s, may depend for completion on the completion of its (consequent) successor transaction. Dependent transactions and their potential consequent(s) are denoted as dependent/consequent(s). Cycles are not forbidden, but the model by Kramer and Magee does assume that the transactions still complete in bounded time and that deadlocks are avoided. It is also required that the initiator of a dependent transaction is informed of the completion of consequent transactions because otherwise a component can not determine when transactions it initiated have completed and hence when it reaches passive status. Each of these assumptions are reasonable and also valid in our own component model.

The problem with dependent transactions is that the passive status may not be reachable for nodes utilizing dependent transactions. Assume the three nodes in Figure 6 which was taken from [16]. Suppose that node $N3$ is in a passive status and $N1$ has initiated transaction a. In this situation, transaction a can not complete, because b can not complete, because c may not be initiated by $N3$ as it is in a passive status. Consequently, neither $N1$ or $N2$ can move into the passive status in bounded time.

The solution proposed by Kramer and Magee is to generalize the definition of passive status to include the means for dependent transactions to complete:

Definition 6 (Generalized Passive Status) *A node in the generalized passive status must accept and service transactions and* initiate consequent transactions, *but:*

(i) it is not currently engaged in a (non-consequent) transaction that it initiated

(ii) it will not initiate new (non-consequent) transactions

We adopt this solution, but have to cope with an extra problem because in our model, components are not aware of their participation in transactions they did not initiate. It is therefore not possible for a component to identify those transactions that are consequent, and those that are not. Adding this information to the component code is unacceptable however, as this would increase implicit coupling between components and strongly hinder reuse. In DRACO , this problem is solved by performing additional bookkeeping when messages are send or received [17]. Whenever a component sends out messages in the context of a transaction it initiates, it tags these messages. The DRACO message delivery system recognizes these tags and transparently forwards the tag to all messages send out as part of that transaction.

As a first step to place a component C in a quiescent status, the LUMconveniently makes use of this feature when it composes a set of ongoing transactions that must finish before quiescence can be achieved. To do so, the LUM queries all the adjacent components of C, checks which ongoing transactions involve C as a participant, and stores them in the `Initial List`.

Algorithm 1 Receive(m)

```
if tag($m$) part of InitialList then
    $messageThread \leftarrow$ current thread
    $outId \leftarrow 0$
    $struct \leftarrow <messageThread, C,$ tag($m$), $outId>$
    Execute($m$)
    Rebuild InitialList
    if InitialList is empty then
        Quiescence Reached
    end if
else //Not part of an ongoing transaction
    Queue($m$) at the deliveryMessageHandler
end if
```

Whenever a message is received by C or by one of its adjacent components, their message handler intercepts that message and checks wether the message is a part of a transaction of the `Initial List`. If so, the message handler stores a tuple containing the ID of the current thread, the ID of the current component, the tag of the received message and a new tag for the outgoing message. Afterwards it executes the message, and checks wether the `Initial List` is empty, which would mean quiescence is reached for C. If the message is not part of a transaction in the `Initial List`, it is queued, as its execution is not necessary to reach quiescence for C.

Algorithm 2 Send(m)

```
$sendThread \leftarrow$ current thread
$struct \leftarrow$ LUM.get($sendThread$)
if exists($struct$) then
    S.outId++
    tag(m) $\leftarrow S.tag$ + "." + $S.N$ + $S.outId$
end if
messageHandler.deliver($m$)
```

When a message is sent by C or one of its adjacent components, their message handler intercepts that message,

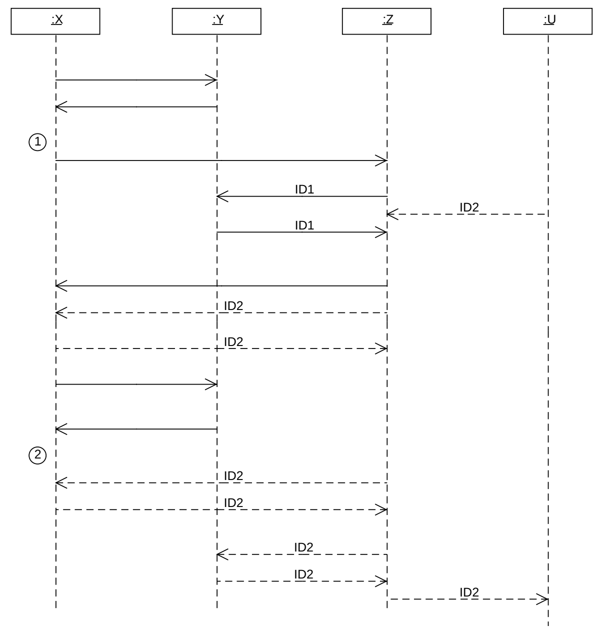


Figure 7. 2-phase optimization for achieving quiescence.

looks for the tuple that corresponds with the current thread ID and uses this information as a base for a new tag that correctly identifies the message as part of the transaction.

As soon as all transactions in the initial list have ended, only tagged messages are delivered to the component. Since all transactions are assumed to be bound in time, this moment is certain to occur in bounded time. In the example on Figure 7, this moment is identified by label 2. Because all other messages are queued by the delivery message handlers, ongoing transactions can complete, but no new transactions can be initiated.

The above implementation assumes that all messages belonging to a transaction are tagged so that when the `InitialList` is constructed, the tags of active transactions can be retrieved. It is possible to relax that requirement to minimize overhead during normal application execution. If components are still aware of transactions they initiated, but the messages that belong to these transaction and resulting dependent transactions are not tagged, the LUMcan still achieve quiescence, albeit using a slightly more complex algorithm consisting of two phases (Figure 7). In the first phase, all messages are delivered to their destination because it is not known to which transaction each message belongs. In addition, the LUMstarts tagging all newly initiated transactions that involve C. The second phase starts as soon as the `InitialList` is empty. At this moment, identified by label 2, we have the situation assumed in the basic algorithm described above since all ongoing transactions involving C are now tagged.

7 Related work

While we are weakening the notion of quiescence, there have been others that implemented the notion of quiescence as presented in [16]. [5] shows an implementation of the quiescence model for distributed component systems. [8] presents resource aware components; components that are aware of their environment, and can react upon changes in that environment. Taking this a further brings us to the world of ambient software, where components adapt their behavior using information from the outer world [22]. Those approaches however, consistently break the black-box design principle, lowering reusability.

The notion of tranquility is not only applicable in component systems [19, 18], but also in all other paradigms that allow modularization and explicit interaction. We have found applications of the notion of quiescence in procedural programming, service-oriented programming and in object-oriented systems. In all these approaches, the notion of tranquility could be introduced for enhancing the dynamic updatability.

As early as 1976, Fabry presented a system allowing for dynamic changes of abstract data types written in procedural languages [6]. Other systems in this area were developed by Gupta [11], Hicks [13] and Hoffmeister. The latter even stated in [14] that the notion of quiescence is too strong and that entities can be safe without enforcing quiescence. Her approach still does not offer support for black box entities however.

Service-oriented systems [3] are decomposed into different entities which are providing and requesting services. Two entities are connected by service contracts [7]. This explicit linking of entities enhances the decoupling. This explains why [15] introduces the notion of quiescence for service-oriented system.

Our approach is also applicable to ordinary object-oriented systems. The Dynamically Alterable System (DAS) [9] is an operating system from the late seventies. It supports the replacement of an object by another one with the same interface. In the DAS system, the in- and out- operations on objects are first class, ensuring less coupling between the objects and allowing data-restructuring. In [12], Gupta et al show how object-oriented systems should be updated dynamically. They also claim that the programs should be in a quiescent status before the updates should be carried out.

8 Conclusion

This paper addresses the problem of state consistency before and after a dynamic change. The problem was originally identified by Kramer and Magee [16], who introduced the notion of quiescence as a sufficient and necessary condition to ensure state consistency. Although they have proved that quiescence is reachable and sufficient for ensuring state consistency, their approach suffers from two major drawbacks. Firstly, quiescence causes serious disruption in the application which is updated due to the large number of nodes that need to be passivated. Secondly, their approach breaks the principle of black-box design because their nodes must not only be aware of the services that they are providing, but also about the nodes that are requesting a service. This clearly increases the coupling between the involved components, and hinders reusability.

In this paper, we have overcome these two drawbacks by using the notion of tranquility. Although not guaranteed to be reachable, tranquility can be achieved quickly in the majority of cases. We have shown that tranquility – when reached – is a sufficient condition for ensuring state consistency. In the few cases that tranquility can not be reached in bounded time, a fallback mechanism to quiescence can easily be implemented.

The advantages of tranquility over quiescence are threefold. First, tranquility has a much smaller change impact than quiescence, since a node in a tranquil status does not require all of its adjacent nodes to be in a passive status. Second, tranquility exploits properties of the black-box design of system nodes, since dependent transactions are considered to be independent from each other. Third, tranquility allows the replacement of nodes at times when it is semantically correct to do so, but when the quiescence condition does not hold.

We have shown that tranquility is easy to implement without breaking the black-box design of each of the system nodes. All necessary information can be retrieved from a detailed specification of the transaction supplied by the initiator. When fallback to quiescence is required, transparent message tagging can be used in order to avoid breaking the black-box nature of nodes.

References

[1] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. *Lecture Notes in Computer Science: ECOOP 2003 - Object Oriented Programming*, 2743:74–102, 2003.

[2] Y. Berbers, P. Rigole, Y. Vandewoude, and S. V. Baelen. Components and contracts in software development for embedded systems. In *Proceedings of the first European Conference on the Use of Modern Information and Communication Technologies*, pages 219–226, 2004.

[3] G. Bieber and J. Carpenter. Introduction to service-oriented programming. `www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf`, September 2001.

[4] S. Bohner and R. Arnold. An introduction to software change impact analysis. *IEEE Computer Society Press*, pages 1–26, 1996.

[5] A. L. de Moura, C. Ururahy, R. Cerque, and N. Rodriguez. Dynamic support for distributed auto-adaptive applications. In M. Akşit and Z. Choukair, editors, *Proc. 2nd Int'l Workshop on Aspect Oriented Programming for Distributed Computing Systems (ICDCS-2002), Vol. 2*, 2002.

[6] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd ICSE*, pages 470–476, San Francisco, CA, USA, 1976.

[7] P. Gahide, N. Bouraqadi, and L. Duchien. Promoting component reuse by integrating aspects and contracts in an architecture model. In *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, 2002.

[8] J. Gerlach and S. V. Baelen. Run-time evolution and dynamic (re)configuration of components: Model, notation, process and system support. Technical report, Katholieke Universiteit Leuven, 2003.

[9] H. Goullon, R. Isle, and K.-P. LShr. Dynamic restructuring in an experimental operating system. In *International Conference on Software Engineering*, 1978.

[10] J. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Morgan Kauffman, 1993.

[11] D. Gupta. *On-line Software Version Change.* PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, Nobember 1994.

[12] D. Gupta, P. Jalote, and G. Barua. A formal framework for online software version change. In *IEEE Transactions on Software Engineering*, pages 120–131, February 1996.

[13] M. Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, 2001.

[14] C. R. Hofmeister. *Dynamic reconfiguration of distributed applications*. PhD thesis, University of Maryland, College Park, MD 20742, 1993.

[15] A. Ketfi and N. Belkhatir. Dynamic interface adaptability in service oriented software. In *Workshop on Component-Oriented Programming*, 2003.

[16] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[17] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM Press.

[18] A. Rausch. Software evolution in componentware - a practical approach. In *Proc. of the Australian Software Engineering Conference*, 2000.

[19] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, January 1998.

[20] D. Urting, S. V. Baelen, T. Holvoet, P. Rigole, Y. Vandewoude, and Y. Berbers. A tool for component based design of embedded software. In *Proceedings of Tools Pacific 2002*, Februari 2002.

[21] Y. Vandewoude and Y. Berbers. Semantically sane component preemption. In *Proceedings of ERCIM06*, April 2006.

[22] A. Wils, P. Rigole, Y. Berbers, and K. D. Vlaminck. Ambient computing using component resource contracts. In *Advances in Computer Science and Technology*, 2004.