Two Paradigms of Composition

Ashley McNeile
Metamaxim Ltd.
48 Brunswick Gardens
London, U.K.
ashley.mcneile@metamaxim.com

ABSTRACT

We use a small example to discuss how two different formal modeling languages address the interaction between data and behavior using parallel composition. We use this discussion to highlight the distinction between *synchronous* and *asynchronous* semantics of parallel composition, a distinction not hitherto properly discussed in the context defining the interaction between behavior and global or shared data. We discuss some uses of parallel composition in systems engineering, and some considerations that determine whether synchronous or asynchronous semantics are best aligned to these uses.

Categories and Subject Descriptors

F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Interactive and reactive computation, Parallelism and concurrency*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*State diagrams*

General Terms

Behavior Modeling, Concurrency, Composition

Keywords

parallel composition, process algebra, labeled transition systems, semantics, synchronous reactive languages

1. INTRODUCTION

This paper is concerned with modeling computation where data and behavior interact, and how *parallel composition* techniques, as have been developed and explored widely in the context of process algebras such as CSP, CCS and ACP, are used to model such interaction. Our aim is to highlight some issues in the definition of the semantics of parallel composition that have significant impact on when and how different languages should be used. In particular, we show that there is a distinction to be made between *synchronous* and *asynchronous* parallel composition, reflecting the degree of concurrency assumed to be possible between composed behavioral abstractions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
BM-FA Workshop 6 June 2011 Birmingham, UK.
Copyright © 2011 ACM 978-1-4503-0617-1/11/06 ...$10.00.

2. THE BANKING EXAMPLE

The example that we shall use as a basis for the discussion is this:

A certain bank allows its customers to open and maintain any number of separate accounts, but imposes the rule that a customer may only withdraw funds from an account if the result of the withdraw is that she remains in credit overall, where "overall" means across all of her accounts.

This example is chosen because it is easy to understand but is also sufficiently rich to raise interesting questions concerning the way in which data based constraints are modeled using compositional techniques. The obvious interesting feature of the problem is that the constraint on doing a withdraw from one of a customer's accounts is "non-local", as it requires knowledge of the state across the other accounts held by the customer. The interesting question is this: all these accounts clearly exist in *parallel*, but to what extent are they *concurrent*?

2.1 Structure of this Paper

We describe two different models of the this example, in two different composition based languages: *Protocol Modeling* and *mCRL2.* We then:

1. Discuss the differences between these two models of the bank, and how these are driven by the different semantics of the two languages used.
2. Discuss the different software engineering principles that have shaped the semantics of the two languages.
3. Examine how the distinction between synchronous and asynchronous composition has been has been addressed in work on process algebra, and argue that it needs to be re-examined.
4. Take a fresh look at the uses of behavior composition in the software development and verification process.

2.2 Validity of Assessment

Inevitably in papers of this sort there is a danger of using modeling languages inappropriately and hence abusing or misrepresenting their features or capabilities. In extreme

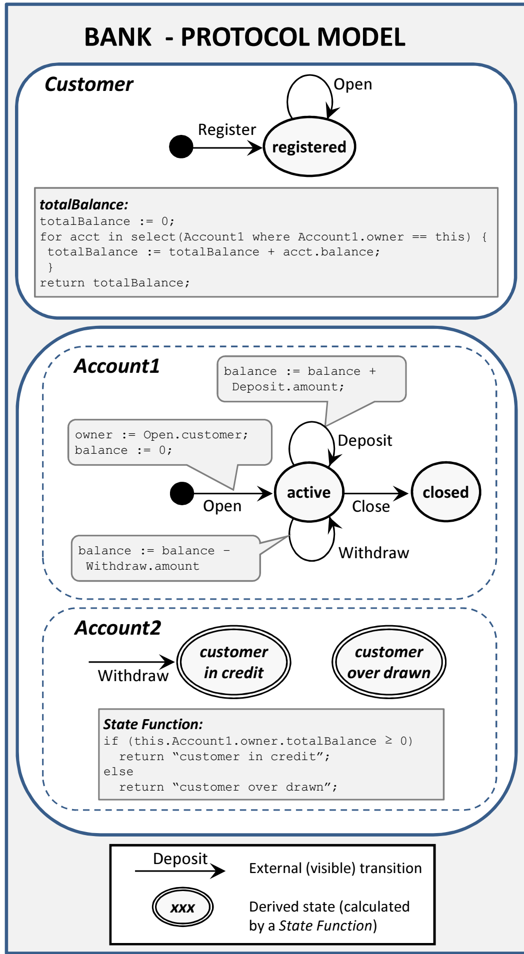

Figure 1: Bank Protocol Model

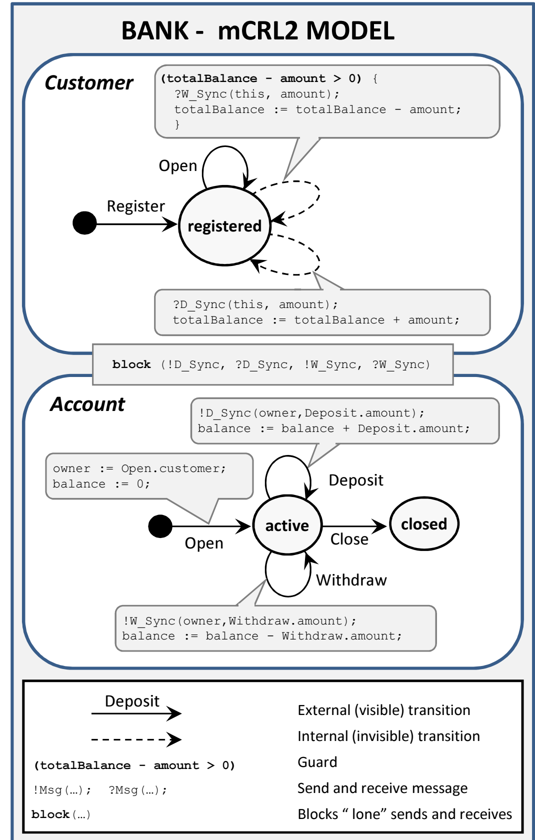

Figure 2: Bank mCRL2 Model

cases this can invalidate the conclusions reached in a comparison of the type this paper attempts. However, we believe that this is not an issue here for the following reasons:

- The Protocol Model solution was developed by the author, who is also author of the Protocol Modeling technique.
- The mCRL2 model is based on a solution provided by Michel Reniers[1] who has been involved in the development of both mCRL2 and of ACP, the process algebra on which it is based.

3. THE TWO MODELS

In this section we describe the two models of the banking example given in Section 2. Our interest is in how the models address behavioral constraints, and to focus on this both models present a very simplified view of banking: account behavior is confined to deposits and withdraws, we do not model processes concerned with account opening or closure, and we ignore such issues of security and authorization.

Sources for the two models can be downloaded from: `http://www.metamaxim.com/download/models/paradigms.zip`.

[1]M.A. Reniers, Assistant Professor, Technische Universiteit Eindhoven.

3.1 Protocol Model

The first language is *Protocol Modeling*. This is a state-transition based modeling technique that uses ideas borrowed from process algebras to describe event based systems that have rich interaction between data and behavior. A full description is given in McNeile and Simons [3]. Using Protocol Modeling, a software system is modeled as a set of machines called *protocol machines* which work together in composition to ensure that the behavioral constraints are met. Protocol machines have behavior and data, and are conceptually like "objects" in an Object Oriented paradigm; so machines own data and only a machine that owns an item of data may alter its value. Unlike some other compositional techniques (such as mCRL2, the other we use in this paper) one machine in a Protocol Model is allowed to access the data owned by other, composed, machines.

A Protocol Model for the banking example is shown in Figure 1. The model consists of two "entities": *Customer* and *Account*. *Customer* is described using a single machine shown at the top of Figure 1 and *Account* by the composed machines *Account1* and *Account2* shown in the lower part of the figure. These work together as follows:

- The *Customer* machine ensures that an *Account* may only be opened for a customer who is in the state `registered`. However, once registered, any number of accounts may be opened for that customer.
- The box below the state diagram shows that customer has a derived attribute called *totalBalance*. This is

the sum of the balances of the accounts owned by this customer. These accounts are selected based on an attribute *owner* in the account, a "foreign key" that points to the owning customer.

- *Account1* reflects the familiar account lifecycle. The bubbles attached to the transitions show how the stored attributes of this machines are updated. With the *Open* event the *owner* is stored (using the customer identifier carried in the *Open* event), and the *balance* initialized. The *balance* is updated by *Deposit* and *Withdraw* events (using the *amount* attribute carried in the event) in the obvious way.

- *Account2* enforces the constraint that no *Withdraw* event can take the total balance of the customer below zero. This machine has a derived state, meaning that the state is calculated on demand by a *state function* shown below the diagram for *Account2*. The state function uses the *totalBalance* of the *owner* of the account to distinguish between the customer being in credit and being over drawn. The semantics of the transition with its ending state as `customer in credit` is that this state must pertain after a *Withdraw* has taken place.

Note that *Withdraw* is subject to the constraints of both *Account1* and *Account2*, so that a withdraw can only happen if both *Account1* is in the state `active` and *Account2* ends up in state `customer in credit`. If either *Account1* or *Account2* does not allow the event, the event cannot take place (is refused). The composition is therefore in line with the || composition of CSP, whereby an event being refused by any component of a composition gives refusal by the composition.

Instantiation of the model entails instantiation of a *Customer* machine for each customer who registers with the bank, and instantiation of an *Account1* + *Account2* pair for each account opened. All the machines instantiated are composed in parallel.

3.2 mCRL2 Model

The mCRL2 language supports modeling the behavior of behavioral entities that run in parallel and interact with each other. The language has its theoretical basis in the process algebra ACP (Algebra of Communicating Processes) [7]. An mCRL2 model for the bank example is presented graphically in Figure 2 for comparison with the Protocol Model. This figure renders the mCRL2 model in graphical form, and this has required some interpretation of the language constructs, for instance using topological loops to represent recursion. But this does not change the essentials of the solution.

Like the Protocol Model, the mCRL2 model consists of two "entities": *Customer* described by the machine shown in the upper part of Figure 2 and *Account* described by machine shown in the lower part of the figure. The basic state/transition topology of the *Customer* and *Account* machines are the same as the *Customer* and *Account1* machines in the Protocol Model. This is to be expected, as both models represent the same situation. However, the way in which the behavioral constraint on withdraws is treated is quite different. The basic idea in the mCRL2 model is that the customer machine maintains a total balance of all the accounts it owns, and uses this to "give permission" for a *Withdraw*. This means that the *Customer* machine has to participate in any event that changes the balance of any account it owns. This participation is achieved using messages, *D_Sync* and *W_Sync*, that an account machine writes to its owning customer machine in the context of a *Deposit* or *Withdraw* respectively. Using a *guard*, the customer machine can refuse to receive a *W_Sync* and hence prevent an offending *Withdraw* event. More specifically:

- The `block` statement (shown between the two state diagrams) says that send and receive actions cannot happen by themselves. This makes the communications between *Account* and *Customer* synchronous.

- The *Account* machine sends a message as a part of handling a *Deposit* or *Withdraw*, telling the owning customer of the amount that has been deposited or withdrawn.

- The *Customer* machine receives the message and updates its *totalBalance*. But the receive of the *W_Sync* has a **guard** requiring that the resultant *totalBalance* be positive.

- As the `block` requires the send and receive to happen together, if the guard evaluates to false so that the receive of the *W_Sync* message cannot happen, then the send cannot happen either. The event *Withdraw* in *Account* that caused the send is thereby disallowed.

As with the Protocol Model, instantiation of the mCRL2 model entails instantiation of a *Customer* machine for each customer who registers with the bank, and instantiation of an *Account* machine for each account opened. All the machines instantiated are composed in parallel.

4. DISCUSSION

An obvious difference between these two models is the fact that the mCRL2 model maintains a total balance in the customer machine whereas the Protocol Model does not, as it calculates the total balance on-the-fly from the underlying accounts. This difference is **not** an artefact of the way in which we have chosen to build the two models, but is dictated by the underlying principles on which the languages are predicated, and which appear to be in conflict. A key aim of this paper is to isolate and understand this conflict.

Protocol Modeling supports a compositional style of modeling and aims to allow the construction of models that combine:

- Economy and simplicity of representation, with

- Encapsulation of behavior.

The first (economy and simplicity of representation) refers to the idea that the set facts stored in a model should be mutually independent, analogous to a *basis* of a vector space[2]: so that no one stored fact in a model can be computed from others. This is an idea that, although it has no formal basis in Computer Science, has emerged as a principle of software engineering known by (among other terms) the DRY ("Don't Repeat Yourself") Principle.[3] This principle distills

[2]See `http://en.wikipedia.org/wiki/Basis_(linear_algebra)`
[3]See `http://en.wikipedia.org/wiki/Don't_repeat_yourself`

accumulated experience in building models where data and behavior interact, which indicates that models which contain redundancy tend to be more complex, fragile and difficult to amend than models that do not. Keeping a *total balance* at the customer level is not compatible with this principle, as it duplicates the information in the individual accounts and so, in principle, could be re-established if it were to be lost by adding the account balances together. The Protocol Model solution uses a derived attribute, thereby conforming to the DRY principle. An insight into the wisdom of the DRY principle can be obtained by considering the following change: Suppose that it is required to add a new event to the model that transfers an account from one customer to another. In the Protocol Model a new event, *Change Owner*, can be added to *Account1* that changes the value of the *owner* attribute to point to a different customer. Because the *Customer* machine contains no information about the owned accounts, no change is required to this machine. Making the same change to the mCRL2 model is more complex, as the *totalBalance* attributes of both the old and the new owners of the account would have to be adjusted to reflect the change, so the customer level machines must also engage in the event.

The second aim (encapsulation of behavior) is to capture the *protocol of an entity* locally, within the definition of that entity. *Withdraw* is in the protocol of account and not of the protocol of customer, so encapsulation dictates that the rules constraining withdraws should belong to the account entity. In the the Protocol Model the protocol for *Withdraw* is appropriately located in *Account1* and *Account2*, conforming to encapsulation; but in the mCRL2 model part of this protocol is encoded in the guard in *Customer*.

The mCRL2 language follows thinking that has emerged from work on process algebras, which concludes that sharing of data is not compatible with the ability to understand or reason about the behavior when parallel composition is involved. For instance, in their book on ACP (which is the theoretical basis for mCRL2) Baeten et al. [7] state that:

The independent execution of parallel processes makes it difficult or impossible to determine the values of global variables at any given moment. It turns out to be simpler to let each process have its own local variables, and to denote exchange of information explicitly via message passing.

As a result, the mCRL2 language provides no means whereby one process can directly access data owned by another; so when data in one process is needed by another it has to be passed explicitly in a message. The need to maintain a total balance at the customer level then becomes unavoidable, as establishing the customer balance (and ensuring it remains in credit) would require a synchronous message interaction across the set of accounts owned by a customer, which is not possible as the membership of this set is not statically defined in the model.

The divergence of thinking that these two solutions reveal can be understood in terms of two related topics:

- Distinctions in the *semantics of composition*, and
- Consideration of different *roles that modeling plays* in software development.

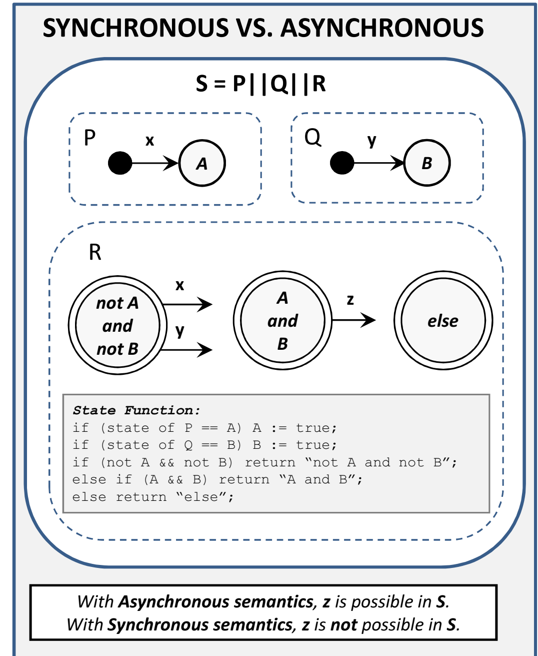

Figure 3: Synchronous vs. Asynchronous

We discuss these in turn in the following two sections.

5. SEMANTICS OF COMPOSITION

Suppose that the machines that represent a customer's accounts are distributed, so that each account instance is hosted by a different processor. It could be that, at all times, at least one of the account machines owned by a customer is in mid execution of some local transaction and so unable to contribute a coherent current balance to the customer total. One would be correct in supposing that this level of concurrency is not compatible with the notion, entailed in the Protocol Model shown in Figure 1, that the customer machine can derive, at will, the total balance of all its accounts. The degree of concurrency yielded by this distribution gives rise to *asynchronous composition*, whereas the semantics of the Protocol Model language is based on *synchronous composition*.

5.1 Illustration of Semantics

The difference between these two interpretations of composition can be seen informally in the example shown in Figure 3. This figure shows a model with a machine S that consists of three machines, $\{P, Q, R\}$, composed using $\|$. We take the $\|$ operator to have the CSP meaning, so that an action is refused by S if refused by at least one component. We suppose that P and Q start at their respective `black dots` and therefore R starts in the state `not A and not B`. This means that P allows x, Q allows y, and R allows both x and y. Under the rules of $\|$ composition, S therefore allows x and y. Now consider how the execution of S advances under two different assumptions about the behavioral semantics, *synchronous* and *asynchronous*:

Synchronous Composition.

In the synchronous case, S advances one action at a time.

CSP Conditional Operators

CSP [4] includes a mechanism to assign values to variables and a *conditional operator* whereby the assigned values of variables can influence behavior.

$$(a \rightarrow P \not\prec \texttt{cond} \not\succ b \rightarrow P) \tag{1}$$

In (1) the process expression to the right (action a followed by P) is chosen if `cond` is true, otherwise the process expression to the left is chosen. However, Hoare notes that *to deal effectively with assignment in concurrent processes, it is necessary to impose a restriction that no variable assigned in one concurrent process can ever be used in another.*

ACP Guarded Commands

ACP [7] includes a concept called a *guarded command*, of the form $\Phi :\rightarrow P$, with the intuitive meaning '*if* Φ *then* P'. This is clearly similar to the conditional operator of CSP, in that the CSP expression in (1) can be expressed in ACP as $(\texttt{cond} :\rightarrow P \;+\; \neg\texttt{cond} :\rightarrow Q)$. However, there is no stipulation in ACP, as there is in CSP, that a propositional variable such as Φ is local to a process: it can be given a value in one and used in another.

The ACP authors note that, because a propositional variable (such as Φ) in one process can be changed by a composed process it is necessary to consider *the behavior [of a process] for all possible valuations of the propositional variables in all states that may occur during the execution of the process.*

LOTOS with Global Variables

LOTOS (*Language of Temporal Ordering Specification*) is a process algebra based on early versions of CCS and CSP. In 1995, Khoumsi and Bochmann explored the possibility of introducing support for global variables into LOTOS [1]. The approach is to encase each action, σ, in a *transaction* $<\theta, \sigma, \phi>$ by adding an *enabling condition*, θ, and an *update function*, ϕ. An action is only enabled if the enabling condition is true (so this is the same as a guard); and execution of the action results also in execution of the update function that gives guards new values.

In order to ensure atomicity of transactions, required to serialize updates to a global variable, the authors employ a superstructure (borrowed from database theory) comprising *locking primitives supporting read and write locks, along with a two phase locking protocol* and *a timestamp based priority mechanism for clearing deadlocked transactions.*

Table 1: Examples of Data Treatment

Suppose x happens. P is then in state `A`, Q is still at its `black dot`. R evaluates its state to `else` (as it is in neither of its other two states). As R refuses all actions of its alphabet $\{x,y,z\}$ when in state `else`, under the rules of $\|$ nothing further can happen in S. An exactly similar argument applies if y happens first. These are the only two possibilities, and in neither is z possible in S.

Asynchronous Composition.
In the asynchronous case, there is no discipline of advancing one action at a time. So x and y could take place concurrently and the first coherent state that R obtains could be `A and B`. In this case, R is now in a state where z is possible. As z is not in the alphabets of P and Q, z is then possible in S.

5.2 Treatment in Process Algebra

Some authors in process algebra have made a similar distinction between synchronous and asynchronous composition, notably:

- Bergstra and Klop with their formulation of *ASP* [8], a synchronous variant of the (asynchronous) ACP algebra. They use the term *synchronous co-operation* for the composition used in ASP and *asynchronous co-operation* for that in ACP [9].
- Milner with his formulation of *SCCS* [12], an elegant synchronous variant of his (asynchronous) CCS.

Both of these algebras model synchronous behavior by requiring that all processes in a composition must engage in every step, so it is not possible for a process to execute an action independently of its peers. This means that a composite progresses as though to a clock, with every component process performing exactly one action to every tick. It is important to point out, as Bergstra et al. do, that this distinction is different from that between synchronous and asynchronous *communication*, which concerns whether or not the send and receive actions on a message happen simultaneously.

Milner's SCCS includes an *idle action* which allows a process to engage in a step of a synchronous composition in a "silent" fashion. He goes on to point out that if the processes in a composition can idle repeatedly, the synchronous composition mimics asynchronicity, as a constantly idling process effectively disengages allowing others to proceed independently. Using this idea Milner builds an asynchronous calculus, *ASCCS*, from SCCS and goes on to show that it is possible to encode the original (asynchronous) CCS within ASCCS. Perhaps because this suggests that asynchronous composition is "more general", as it can be regarded as equivalent to a synchronous model relaxed to allow idling; but more particularly because the study of distributed, and therefore necessarily asynchronous, software has provided the prime focus and motivation for research using process algebras, a consensus in the process algebra community has solidified around accepting the primacy of asynchronous semantics, which has become a de-facto standard.

5.3 Adding Data to Process Algebra

Most authors of process algebra have attempted to augment their action based semantic foundations with a facility to model data and the interaction between data and behavior. Different authors have taken different approaches and Table 1 shows three examples. All start from the assumption of *asynchronous semantics*. However this assumption makes it hard or impossible to add a useful concept of shared or global data, allowing the way data interacts with behavior to be modeled. Looking at the three examples in Table 1:

- In the CSP approach, one process cannot use a value set by another, so data must be completely local and not shared across composed processes. CSP therefore has no notion of shared data and the only means for passing data between composed processes is via messages. The CSP conditional operator construct could not therefore be used to model the example in Figure 3 as R could not base its behavior on changes of state within P or Q.

- The ACP approach has a concept of shared data in its guard construct. However all possible combinations of data values that could occur in any execution of the composed processes must be allowed. If we apply this to the example in Figure 3 we have to assume that R could see any combination of $\{\texttt{A}, \texttt{not A}\} \times \{\texttt{B}, \texttt{not B}\}$, and in particular the combination `A and B`. This is not strong enough to preclude the action z in Figure 3, nor to specify the constraint on withdrawals required in the banking example.

- In the LOTOS plus global data approach, transactions can be used to coerce the underlying asynchronous language into behaving synchronously. This results in a hybrid synchronous/asynchronous language whose semantics has no simple mathematical denotation and has significant limits to its expressive power. The model in Figure 3 could not be represented, as the language prohibits composition of two processes with a common action where the enabling condition of the action in one is altered by the update function of the same action in the other, as is the case with action x in P and R.

The challenge of adding a useful of concept of shared data to an asynchronous model can be further illustrated by considering a small amendment to the model in Figure 3, as follows. Suppose that instead of a single x message, P receives a stream of x messages and cycles between states `A` and `A'`; and Q similarly receives a stream of y messages and cycles between `B` and `B'`. With asynchronous semantics it could be that, whenever R tries to establish its own state by obtaining the states of P and Q, either P is incoherent (in transit between `A` and `A'`) or Q is incoherent, or both. In this case R's state could be permanently incoherent, and so the behavior of S is permanently undefined. We could escape this problem by supposing that if R seeks the state of P it obtains the last coherent state that pertained in P, even if P has now moved on from that state (and similarly for Q). As noted above in the context if ACP's guarded commands, which exhibit the same difficulty, this is too weak to enable useful specification of behavior such as that in the banking example introduced earlier.

These examples help explain why, as noted in Section 4, Baeten et al. eschew global variables in favor of message passing, and why mCRL2 provides no means for composed processes to share data.

5.4 Synchronous Composition Revisited

The picture changes if the use of globally shared data is considered in the context of **synchronous** rather than, as above, **asynchronous** composition. With inter-process access allowed (as R accesses the states of P and Q in Figure 3) whether a composed process is mimicking independence with silent idle steps or is truly asynchronous becomes substantive, as the former affords synchronization points where coherent data or state may be read from another process and the latter does not. For this reason we think that synchronous and asynchronous composition be recognized as two different *paradigms of composition*, with:

- The asynchronous paradigm based on true autonomy, not supporting shared data (so data is exchanged by message passing).

- The synchronous paradigm based on stepwise synchronicity, allowing data be shared to support rich interaction between data and behavior in a direct fashion.

Both paradigms of composition admit clean mathematical denotation (discussion of the formal semantics is beyond the scope of this paper); and we argue in the next section that both have a role to play in design and verification of software, but that these roles are different.

6. THE ROLES OF MODELING

One might expect that the two compositional paradigms are both useful, but in different contexts. We now argue this case, and the key determinant of which is appropriate is the source of the parallelism that motivates the use of composition. There are two drivers for the use of composition in the description or specification of software: parallelism that derives from the *implementation* and parallelism that derives from the *problem*.[4] These are essentially independent, in that a given software system may entail either one, neither or both. We look at each in turn.

6.1 Parallelism in the Implementation

Here we are concerned with software that is to be distributed across multiple processors/threads. Such distribution may be needed for reasons of performance, fault tolerance, or to meet physical distribution requirements. In this case separate definition of the distributed parts is essential, as each processor/thread has to be given an autonomous source of behavioral instruction to obey. The composition results from the parallel execution of the parts and their interaction using message passing and/or access to shared memory.

Because the processors/threads used for implementation are conceptually asynchronous, it is clear that asynchronous composition is the applicable paradigm. As expected, this is the domain where languages based on asynchronous semantics are strong. The typical use of languages such as

[4]Some authors refer to the distinction as *physical* versus *logical* concurrency. See, for instance, Halbwachs [11].

mCRL2 is to analyze the behavior of distributed systems, to ensure that the concurrency does not allow behaviors that are not expected or are pathological (such as error or deadlock). This is done by using analysis tools that generate, and selectively or exhaustively explore, the state space of the behavior.

6.2 Parallelism in the Problem

Here we are concerned with problems that lend themselves to description using composition, whether the implementation will be distributed or not. Two notable cases are:

- The software concerns an external domain populated by independent but interconnected entities. Such applications are based on an *object model*, and the software instantiates objects that represent (are *analogues* for) the real world entities which the application is required to track and control.[5]

- The software models the state and progress of a business process which, because different activities within the overall process can be undertaken in parallel by different parties, is naturally represented using a composition of parallel streams.

The concern here is to construct a model which captures the behavioral **requirement**; where the requirement reflects *what the system is about and what it has to do* and is not concerned (in general) with *how it is to be implemented.* Moreover, the concern is to achieve a description that achieves *the greatest possible clarity and simplicity*, to maximize assurance that the software built to realize the behavior is fit for the purpose for which it is intended and can be easily amended as requirements change. Generally speaking, the development of a model that represents behavioral requirements will be iterative and incremental. It is important that the medium used for modeling supports the process, and in this regard the following are key:

Reasoning.
It should be easy to reason about the the behavior of the model. Once intellectual control over a model is lost, so is the assurance of its correctness as a representation of the requirements. Reasoning about a synchronous model is generally much easier than reasoning about an asynchronous model, for the obvious reason that it is not necessary to think about the possibility of multiple events happening at the same time. Good encapsulation of behavior helps too, as without it the specification of the protocol of an entity (the rules that determine when actions for that entity are possible and when they are not) becomes distributed through the model, so cannot be understood just from the model of that entity.

Evolution.
It should be easy to add features to and remove features from the model. It is generally easier to evolve models that are non-redundant (built to the DRY principle) as such models tend to have lower coupling between their parts. This is because, in a redundant model, the machinery required to maintain consistency generally creates coupling. For instance, functionality to allow transfers of funds between accounts is easier in the Protocol Model of the bank, as there is no need to consider any changes at the customer level associated with maintenance of a total balance.

Refactoring.
As a model evolves so its structure and clarity tend to degrade and need to be re-established by refactoring. This is much easier with synchronous semantics. Suppose that a behavioral machine M is refactored into $M' \parallel M''$ because the behavior is more clearly represented this way. If the $\parallel$ operator is assumed to have asynchronous semantics then we would have to manage any data access between M' and M'' using messages passed between the two. This mitigates against the factorizing of complex behavior into a composition of simple parts, or of rearranging such a factorization once made, which is normally key to successful refactoring.

Simplicity.
The simplest solution is to be preferred where it meets the requirements. As synchronous behavior is a special case of asynchronous, it is always the case that the simplest solution will be one that works in a synchronous context.

These arguments contribute to a convincing case that synchronous language semantics are best able to support the processes involved in creating models to represent problem driven parallelism.

6.3 Parallelism in Both

The forgoing discussion suggests that we need to separate the two concerns of *problem* and *implementation* driven parallelism, and use different modeling languages, with different compositional semantics, for each. But a problem of any scale and complexity is going to have both, and this presents the question: How do we use two languages in combination to address the two concerns? Full discussion of this is beyond the scope of this paper, but some ideas are given below.

Suppose that we are starting from a synchronous model that captures the required behavior (including its problem driven parallelism), and is known to be behaviorally correct. To achieve a correct implementation one or more of the following techniques might be used:

- Use a design technique that is known, through formal reasoning, to preserve behavior when a synchronous design is mapped to an asynchronous implementation. An example is the use of *end point projection* to obtain behavior definitions for the participants in an asynchronous collaboration as described in choreography theory, for instance by the author in [2].

- Use techniques, such as database locking and semaphores, that allow synchronous behavior to be protected (and thus preserved) in a distributed implementation and thereby ensure that the behavior of the synchronous model is not broken. For instance, it would be possible to achieve a correct implementation of the Protocol Model of the banking example by making any account transaction (event) subject to a lock placed at the customer level, thereby serializing the events on a given customer.

[5] Jackson uses the term *analogic model* for a software model that shadows a domain in this way. See his book *Problem Frames* [10].

- Use an implementation technique that supports synchronous parallelism but does not employ distribution, for instance by using a *synchronous reactive* language such as Esterel [6].
- Use standardized patterns for handling concurrency in a distributed system, for example the *optimistic concurrency* pattern used to handle multi-user concurrency in on-line transactional systems.

Where a synchronous behavior model is not available, or a synchronous model has been mapped to distributed implementation in a way that is not guaranteed to preserve behavior, we move into the domain that is addressed by asynchronous languages modeling languages such as mCRL2. This is the case, for instance, where a GALS (globally asynchronous, locally synchronous) style implementation is used, as described by Potop-Butucaru and Caillaud [5].

7. CONCLUSION

The main points of this paper can be summarized as follows:

1. There are two distinct kinds of compositional paradigm: *synchronous* and *asynchronous*. This distinction becomes formally apparent when considering the handling of *shared or global data* in the context of composition.
2. There is no sensible way to define the handling of shared or global data in the context of the *asynchronous* paradigm. Message passing is the only well defined way for asynchronous components to exchange data.
3. On the other hand, shared data **can** be modeled in the *synchronous* paradigm. This enables compositional modeling of the interaction of data and behavior in a way that is compatible with good practice principles of software engineering: observing behavioral encapsulation and avoiding redundancy of stored information.
4. Both paradigms have their use and place in systems engineering:
    - Synchronous for the modeling of *problem* driven parallelism, and
    - Asynchronous for the modeling of *implementation* driven parallelism (distribution).

We think it is a mistake to hold that “one compositional paradigm fits all”. Instead, both paradigms should be accorded recognition, and the software engineer should choose the one best aligned to the problem at hand. This will enable us to avoid the methodological equivalent of trying to force a screw into a piece of wood with a hammer.

References

[1] A. Khoumsi and G. von Bochmann. Protocol Synthesis using Basic Lotos and Global Variables. In *ICNP '95: Proceedings of the 1995 International Conference on Network Protocols*, pages 126–133. IEEE Computer Society, 1995. ISBN 0-8186-7216-1.

[2] A. McNeile. Protocol Contracts with Application to Choreographed Multiparty Collaborations. *Service Oriented Computing and Applications*, 4(2):109–136, June 2010. ISSN 1863-2386. doi: 10.1007/s11761-010-0060-9.

[3] A. McNeile and N. Simons. Protocol Modelling: A Modelling Approach that supports Reusable Behavioural Abstractions. *Journal of Software and System Modeling*, 5(1):91–107, 2006. doi: http://dx.doi.org/10.1007/s10270-005-0100-7.

[4] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[5] D. Potop-Butucaru and B. Caillaud. Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. *Fundam. Inf.*, 78:131–159, January 2007. ISSN 0169-2968.

[6] G. Berry. The foundations of Esterel. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press. ISBN 0-262-16188-5.

[7] J. Baeten, T. Basten and M. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, New York, NY, USA, 2009. ISBN 0521820499, 9780521820493.

[8] J. Bergstra and J. Klop. Process Algebra for Communication and Mutual Exclusion. Technical report, CS-R8409. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1984.

[9] J. Bergstra, J. Klop and J. Tucker. Process Algebra with Asynchronous Communication Mechanisms. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 76–95, 1985. ISBN 3-540-15670-4.

[10] M. Jackson. *Problem frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-59627-X.

[11] N. Halbwachs. Synchronous Programming of Reactive Systems - A Tutorial and Commented Bibliography. In *In Tenth International Conference on Computer-Aided Verification, CAV'98, Vancouver (B.C.), LNCS 1427*, pages 1–16. Springer Verlag, 1998.

[12] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25(3):267 – 310, 1983. ISSN 0304-3975. doi: DOI: 10.1016/0304-3975(83)90114-7.