MASTER'S THESIS

Using an interaction DSL to prevent races in a distributed system

Smeele, T.

Award date:
2022

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the public portal.

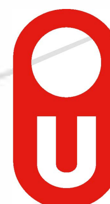
Take down policy
If you believe that this document breaches copyright please contact us at:

pure-support@ou.nl

providing details and we will investigate your claim.

Downloaded from https://research.ou.nl/ on date: 25. Jun. 2024

Open Universiteit
www.ou.nl



Using an interaction DSL to prevent races in a distributed system

Ton Smeele

Student:
Date: 20220401

Open Universiteit

Using an interaction DSL to prevent races in a distributed system

by

Ton Smeele

in partial fulfillment of the requirements for the degree of

Master of Science

in Software Engineering

at the Open University of the Netherlands,
Faculty of Science
Master's Programme in Software Engineering

to be defended publicly on Friday April 1, 2022 at 11:00 AM.

Student number:
Course code: IM9906 SE Graduation Assignment
Thesis committee: Dr. Ir. Sung-Shik Jongmans (supervisor), Open University NL
Dr. Stefano Schivo (reader), Open University NL

Open Universiteit

Contents

List of Figures **iv**

Abstract **vi**

1 Introduction **1**

2 Related work **4**

2.1 Data grids 4

2.1.1 Data grid architecture 4

2.1.2 Coherence challenges in a data grid 5

2.1.3 Data grid requirements 6

2.2 Consistency models. 7

2.3 Multiparty asynchronous session types 9

3 Problem analysis **12**

4 Research scope and methods **14**

4.1 Research questions and methods. 15

4.2 RQ1: Capture the problem case in a session type model 15

4.3 RQ2: Design coherence support as MPST calculus extension 16

4.4 RQ3: Validate the coherence-extended calculus 17

4.5 RQ4: Prototype language extensions that protect coherence 18

4.6 Research validation 18

5 Model the problem case as a protocol **19**

5.1 Problem case modeled as protocol in Scribble 19

5.2 Problem case modeled as LTS in UPPAAL 21
5.3 Prototyping data race avoidance 22
5.4 Validation of problem case model 23

6 An MPST based calculus to support asynchronous data transfers 25
6.1 A calculus for concrete asynchronous interactions 25
6.2 Endpoint perspective: calculus for local language. 27
6.2.1 Local language for a single endpoint 27
6.2.2 Local language for a group of endpoints 28
6.2.3 Local language for a system of endpoints and channels 29
6.3 Endpoint projection 29

7 Coherence model and integration in calculus 31
7.1 Exploring coherence 31
7.2 Coherence modeled as state transition graph 32
7.3 Definition of the coherence property 33
7.4 Calculus extended with coherence model 34
7.4.1 Coherence model as global language extension 34
7.4.2 Coherence as local language extension 34
7.4.3 Endpoint Projection including coherence 35

8 Validation of the coherence-extended calculus 36
8.1 Implementation of calculus in mCRL2 36
8.2 Validation of projection from global to local calculus 38
8.3 Validation of coherence expressiveness 39

9 Prototype language extensions that protect coherence 40
9.1 Towards a strategy for coherence protection 40
9.2 Throughput considerations for coherence 41

9.3 Prototype with coherent update 42

10 Discussion and conclusions **44**

10.1 Discussion 44

10.2 Conclusions 45

10.3 Future work. 45

A Software components used in research **47**

B UPPAAL model of problem use case **48**

C Protocol simulator in mCRL2 **49**

D Coherence property transposed to μ-calculus **51**

E Test cases protocol simulator **52**

F Protocol simulator extensions to support coherent update **53**

Acknowledgements **55**

Bibliography **56**

Glossary of terms and abbreviations **59**

List of Figures

2.1 Data grid architecture. 5

2.2 Processes A and B update x 9

2.3 Other processes read value of x 9

2.4 Global protocol 11

2.5 Projected local protocols 11

3.1 Potential race condition at data object path update. 13

5.1 Scribble protocol for data grid update 20

5.2 UPPAAL template for an Object (Data Object or Replica) 21

5.3 UPPAAL template for AgentNaive 22

5.4 UPPAAL template for AgentMilestone 22

6.1 Structural operational semantics for the global language 26

6.2 Local language structural operational semantics 28

6.3 Structural operational semantics extended for group of endpoints 29

6.4 Structural operational semantics, channels layer 29

7.1 Example execution paths for M, incoherent paths are dashed 32

7.2 Additional layer of global language operational semantics to integrate coherence model 34

7.3 Addition layer of local language operational semantics to integrate coherence model 35

8.1 Protocol simulator overview 36

8.2 Communications between automata, protocol simulator 37

9.1 Chained-coherence with five attributes 41

9.2 Single-chain approach 41

9.3 Two-chain approach 41

9.4 Pseudo code for coherent update operation 42

B.1 Global declarations for problem use case model in UPPAAL 48

C.1 Protocol simulator model, part 1 49

C.2 Protocol simulator model, part 2 50

D.1 Coherence property implementation in μ-calculus 51

E.1 Test cases for protocol simulator 52

F.1 Protocol simulator cohUpd extensions, part 1 53

F.2 Protocol simulator cohUpd extensions, part 2 54

ABSTRACT

Concurrency is becoming increasingly important, as it significantly augments the overall speed of execution in modern parallel and distributed platforms. However, interference from concurrently executing programs may impact the consistency and coherence of persistent data. Whereas locking mechanisms to achieve mutual exclusive access are used to counter this threat in single processor systems, this approach may result in serious performance degradation when applied in distributed systems such as data grids. Therefore commonly deployed consistency models provide a compromise between failure-protection, availability, and consistency, but they do not protect data coherence. We propose a novel method to protect data coherence in non-transactional distributed systems. Our research builds on the multiparty asynchronous session types theory by Honda et al. in which structured conversations between processes are specified as session protocols. We propose to 1) extend the underlying calculus with constructs to support coherence and 2) to enhance a related domain specific language with features that allow software engineers to specify coherence needs. We show that our method can be used to prove that data remains coherent.

1

INTRODUCTION

In the context of computer systems hardware, we commonly assume that unless a system's hardware is malfunctioning, data is guaranteed to remain consistent[1]. For instance, when data is stored in a memory location, subsequent retrievals from that same location should find the data unchanged.

The notion of data consistency is likewise important to software engineering. Software engineers work on the assumption that the value of a data item at any point in a program is solely the result of its initial value and the program instructions subsequently applied to it. Our research is related to the fact that this assumption is not always justified due to interference of other processes.

To protect shared data against consistency issues, commonly a contract is established between processes. The contract facilitates on-demand exclusive access to the shared data for a limited amount of time. Prior to accessing shared data, a process first signals its intent by acquiring an exclusive lock token. If the lock token is in use by another process, it can opt to wait until the token becomes available. Any process that is in possession of the lock may assume that other processes will not concurrently access the shared data, so that it can for instance safely update the data.

This cooperative approach assumes that the lock operations themselves are not interleaved and that processes always release their locks shortly after they have acquired them. For instance in a single processor system non-interleaved locking can be achieved by using cpu instructions that update a register or memory location in an atomic, indivisable, way.

The introduction of distributed system models has forced software engineers to reconsider the implementation of data consistency and associated strategies. A distributed system is a collection of independent computers that appears to its users as a single coherent system [26]. A crucial difference with a central system is the notion that communication be-

[1]The online dictionary Merriam Webster defines consistent as "free from variation or contradiction", see https://www.merriam-webster.com/dictionary/consistent

tween cooperating processes is no longer guaranteed to be near-instant, due to delays inheritantly introduced by network components. For instance a call to lock data might previously have completed in microseconds whereas now it may take a near-second to communicate that message across geographically distributed computers. Lock negotiation across processes suddenly becomes a very time-expensive operation. Do we really need a level of consistency that requires mutually exclusive access to all shared data?

Mutual exclusive access is not always required to keep data consistent. Lamport shows that data consistency is challenged whenever a write operation on a data item is interleaved with one or more read operations on the same item by concurrently running programs [19]. In that case the read operation may return the item's value as it was before the write, or alternatively the value that resulted from the write.

The CAP theorem shows that in a distributed system context a compromise is unavoidable between three commonly desired properties: consistency, availability, and partition-tolerance [11]. Consistency requires data to remain consistent even while the system is responding to concurrent service requests. Services are available when they continue to properly and timely respond to requests under all circumstances. Issues such as network link outages can affect availability. Mutual exclusive access to data resources by one process may affect the progress of another process that attempts to access the same data and as such may impact the availability of the system. Partition-tolerance refers to the ability of services to deal with component failures, for instance when one of the involved processes halts as the result of an exception.

Facing the need for a compromise, instead of solely relying on mutually exclusive access mechanisms, data connection semantics have seen wide adoption [26]. These semantics require a client process to open a connection to the data where it will receive access to a then-current version of the data, possibly as a local copy. Any updates by the client process to the data are applied on this version only. When the client process closes its connection, these updates become visible to any other process that opens a connection to the data thereafter. Hence from a single process perspective, the data appears consistent throughout a connection. Note that scope of data consistency is now limited to this connection: when a process reopens a connection to the same data, the data may turn out to have changed meanwhile.

The result of concurrent updates depends on the implementation. For example distributed file systems typically implement data connection semantics via file open/close operations. Here the convention is that an update from the last data connection closed overwrites the result from an update done by prior connections. Relational database management systems implement data connection semantics via transactions. Only committed data is read by concurrent client processes.[2] The completion of a transaction requires other open transactions to consider the impact of this transaction and where needed reprocess data to take results into account.

Connection semantics fullfill the requirement that a data consistency model must be easy to understand and easy to use for programmers. Obtaining and specifying consistency re-

[2] This is also referenced as multi-versioning concurrency control model with read committed isolation level.

quirements per application can be extremely difficult for software engineers [26]. Therefore software engineers should be facilitated by programming libraries and easy to grasp consistency contracts.

While connection semantics protect the consistency of a single data item, some applications may also require consistency with regard to a *relationship* between data items. Can we manage data coherence with similar ease of use as the single data item consistency? We define data coherence as an invariant content relationship between two or more data items. For example we may require that two related files of a distributed system always have the same content. Note that a connection would only protect the consistency of each file separately. It does not guarantee that either both files are updated or none.

Our research contributes a method for managing data coherence in a non-transactional distributed environment. To this purpose, we define and model data coherence as a property in a distributed process context. Our research builds on the multiparty asynchronous session types methods by Honda et al. that support a formal specification of structured conversations between processes in which properties such as progress are safeguarded [14]. We adapt and extend their calculus to include support for coherence. Our extension facilitates the detection and prevention of coherence issues early in software development cycles.

For practical reasons, we limit our research to coherence requirements of an example non-transactional distributed system architecture: the data grid. Data grids and their specific consistency requirements are introduced in Section 2.1.

The remainder of this document is organized as follows. In Chapter 2, we discuss concepts and prior work related to our research. Chapter 3 details the need for data coherence in a key data grid function and describes the impact of not meeting this property. In Chapter 4, we scope our research in questions, describe the methods we deploy, and we discuss how our research is validated. Chapters 5 through 9 each provide an overview of activities and results related to one of the subquestions of our research. Finally, in Chapter 10 we conclude with a discussion, conclusions, and suggestions for future research.

2

RELATED WORK

In this chapter, we introduce the data grid architecture and a commonly used implementation called iRODS along with architecture specific requirements that include data coherence. Next, we summarize the major categories of consistency models and we review their suitability for our coherence use case. Lastly, we discuss a method to formalize structured interactions between processes called multiparty asynchronous session types. We aim to build on this method to establish data coherence properties.

2.1 DATA GRIDS

The existence of large data collections in distributed systems has prompted the design of an integrated architecture that supports access to and management of distributed data: the *data grid* [7]. In a data grid, data can be located on heterogeneous storage systems and data is replicated for reasons of performance and availability. The grid provides users with easy, uniform access to data. Additionally, in order for data to be found and appraised, the grid maintains contextual information on its data.

2.1.1 DATA GRID ARCHITECTURE

To provide users with easy access to data, data grids virtualize the underlying storage systems. Users do not require any knowledge of the topological network structure of the grid as they reference grid elements by logical names such as *data object* and *resource*. Figure 2.1 shows an application that requests access to a data object called `/PROJ1/DAT5`. Optionally it could have provided an indication of a preferred resource. A resource is a logical name used to reference a particular storage medium on the grid. In our example the available resources are disks `primary` and `secondary`, and tape unit `longterm`. The data grid searches its database for the data object and selects an appropriate file replica of the data object. Subsequently it performs the underlying networking and storage access operations required to access the associated data. In our example the grid selects file `MYDIR/DAT5`

located on storage media `primary` attached to system node `B`.

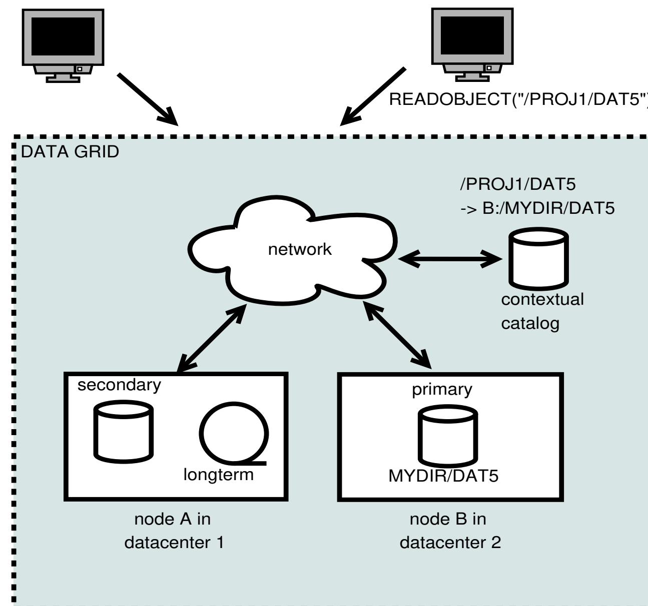


Figure 2.1: Data grid architecture

In data grids, applications annotate data objects with metadata. Metadata is used to find relevant objects based on contextual criteria. As underlying storage systems may have limited and varying capabilities for annotation of data, the metadata is stored separately in a database referred to as the *catalog*.

2.1.2 Coherence challenges in a data grid

Data grid update operations require careful coordination between grid components to support data coherence in a concurrent context. Maintaining the data consistency of an individual data item such as a data object or a file replica is not sufficient. Coherence requirements need to be taken into account as well. For instance when the content of a replica is updated this may also need to be reflected in the related registration of replica file size held in the catalog database.

Should a replica file update fail due to a concurrent operation then the associated catalog update might be undone via a database transaction rollback. The opposite is more difficult to achieve: should a catalog update fail then the file update on storage media might not be as trivial to undo.

Initial data grid architectures would mitigate some of the data coherence risks by limiting replicas to be read-only copies of the data [7]. This constraint has been lifted in more recent implementations such as the iRODS data grid [31].

2.1.3 Data grid requirements

Data grids have specific data consistency and data coherence requirements. Relationships between data items in the same layer (logical or physical) as well as relations between data items that cross these layers, such as the relation between a data object and its replicas, need to remain coherent. Further, existing storage infrastructures need to be integrated seamlessly, data policies may need to be enforced and all these operations must be executed efficiently.

Consistency requirements:

a) Infrastructure compatibility The consistency model must be compatible with components that manage physical data access in the grid. The data grid architecture assumes that, for the purpose of data access performance optimization, existing physical data access mechanisms may need to be shared with high performance compute grid applications [7]. Hence the consistency model must not require any changes to these data grid components.

b) Policy support The consistency model must be compatible with side effects in data grid components. The data grid architecture does not impose restrictions on data policies [31]. For instance, when an application requests the execution of a data operation, this may trigger additional changes to the state of other grid data as well, as the result of a *machine actionable policy*. Alternatively, such policy might use the state of other data as a precondition for an update.

c) Efficient execution The consistency model must facilitate efficient concurrent access to data. The data grid architecture has been designed to meet complex and stringent performance demands while providing access to distributed data collections [7]. Consistency models should avoid to rely on strategies that require exclusive access to data for long time periods as this is likely to negatively impact system progress.

Coherence requirement:

Relation between logical and physical object Virtualized data information needs to remain coherent with physical counterparts. Applications are offered a uniform view on data by the grid and can reference logical data objects for data operations. The data grid must maintain the relationship between these objects and the related physical files [7]. Changes to a logical level property, for instance moving a data object to a different logical location, may imply a need to modify a physical structure, for example move a file to a different directory. During such operations, operations by concurrent processes may not encounter temporarily broken relationships.

In summary, the specific data grid consistency and coherence requirements are a result of its distributed nature, layered data access and its use in environments that demand high performance access to huge amounts of data.

While our research is primarily concerned with the grid's data coherence requirements, the feasibility of implementation options will be influenced by the other requirements such as a need for efficient execution.

2.2 Consistency models

We have defined data coherence as a relation *between* data. Which existing consistency models are suitable candidates for implementing coherence in our data grid use case?

Many data consistency models have been developed for use in non-transactional distributed storage systems. Viotti and Vukolić document over 50 different models, clustered in families of shared characteristics and partially ordered from weak to strong consistency [30]. A weak consistency model provides for little or no data consistency guarantees, a strong model does the opposite.

The model that delivers strongest consistency guarantees is named *linearizable consistency*. It requires that read and write operations of all processes on single data items are executed as if they were atomic operations ordered in time [12]. For example this could be implemented by having each process lock the data prior to each read or write and unlock it immediately after that operation. While this would provide programmers with an easy-to-understand consistency model, the reality of network latency makes linearizability a very difficult if not unattainable target for distributed systems.

Most other models are categorized within one of the families synchronized, staleness, session, fork-based, per-object and causal. Notable exceptions are the eventual and sequential consistency model.

We will briefly review each of these clusters and their relevance in the context of a data grid use case. As selection criteria we use the consistency requirements for a data grid listed in Section 2.1.

Synchronized models provide mechanisms for synchronizing operations on data between processes by defining how and where locks protect these operations [18]. These models can be considered for multiprocessor shared-memory systems. As explained earlier, locking is a less efficient strategy when applied in a distributed systems context such as data grids.

Staleness models guarantee that data written by one process become visible to other processes within a defined limited amount of real-time i.e. seconds [28]. However, real-time guarantees are difficult to establish in data grid systems due to the diversity of supported storage systems. Some storage systems, for instance tape systems, are simply unable to offer access time guarantees.

Session consistency models merely guarantee that data appears consistent in the perspective of a single client process during a stated session with the system [27]. We will refer to this model as *data connection semantics* as we reserve the term session for use with session types. Session types are introduced in Section 2.3. Data connection semantics are less

suitable for our purpose as data coherence requires the consistency guarantees to extend beyond the scope of a single data connection.

Fork-based models seek to counter malicious and erroneous process behavior through algorithms that consider consistency between multiple replicas of data [21]. In contrast, the data grid architecture assumes that underlying storage system components can be trusted by higher level functions.

Per-object consistency models limit the linearizable consistency guarantees to isolated data records or objects [20]. As a result, concurrent access to different objects is not constrained which might considerably improve access performance in a distributed system for some use cases. In a data grid, a suitable scope for an object is difficult to determine as a result of data coherence requirements. For instance, some data grid requests will solely impact a data object and hence would suggest the data object as scope for per-object consistency. However, other requests may impact only one of the data object replica's, while concurrent requests for access to other replica's should remain supported.

Eventual consistency models ensure that after application of all concurrent updates, the replicas of the data will converge to an identical value [27]. With regard to our use case, these models have a similar limitation as session consistency models: distributed processes may each have a different perspective on a persistent data value during the time that the values have not yet converged.

The *sequential consistency* model requires that all processes view the results of data operations in the *same* sequential order, which does not need to be related to the time of occurrence [18]. In this model, the system may process write operations of processes in any order, as long as the order in which values are read by processes is consistent across all processes. As an example of sequential consistency, imagine how a shared logfile is commonly perceived. Several processes might concurrently write data to the log file. The sequence in which data is read from this logfile will appear the same to all processes that concurrently view the log. The sequential consistency model provides the distributed data grid processes with a consistent perspective of the data and therefore is a candidate model for our use case.

Causal consistency seeks to guarantee that if a write operation by process B is potentially influenced by or depends on a write operation from process A then this order is also obeyed by processes that read this data. Hence there is no process in the system that will read what A has written after it has read what B has written [1]. Compared to sequential consistency, the requirements are weaker. Only two writes that are causally related must be read in that order by all processes. The order of other reads can differ between processes.

The *convergent causal* model is similar to the causal consistency model, except that it also requires data to eventually converge to a single value [20].

In summary, data coherence requirements in distributed systems such as data grids could potentially be met using either a (convergent) causal model or a sequential model. A sequential model is more restrictive with regards to the order of events than a causal model. We therefore expect that deployment of a causal model in distributed systems such as data

grids provides for more flexibility and potentially higher runtime throughput than the sequential consistency model although further research is needed to validate this assumption.

To illustrate the causal and sequential models, let's consider a scenario with five concurrently running processes A through E. All processes have access to a shared data item x which is updated by processes A and B as shown in Figure 2.2. The horizontal axis represents the sequence of actions in time. The notation x^1 represents the fact that B reads the value 1 written by A prior to updating the value of x. Figure 2.3 shows values of x read by the other processes C through E at arbitrary points in time.

Process A:	$x^1 := 1$		$x := 3$
Process B:		read x^1; $x := 2$	

Figure 2.2: Processes A and B update x

Process C:	1	3	2	causal
Process D:	1	2	3	causal
Process E:	2	1	3	not causal/seq

Figure 2.3: Other processes read value of x

Does the order in which processes C through E observe the value of x comply with causal and/or sequential consistency properties?

Process B performs a read operation before executing an update operation. Potentially, whatever has been read could have been used as input for this update. Consequently, the process B update operation is now causally related to the operation that most recently updated x prior to B's read operation: the $x := 1$ action performed by process A. Hence to be causally consistent, other processes may not observe the value 1 in a read operation after having read the value 2. Both processes C and D comply with this requirement even while they show a different order for other, unrelated updates. Process E violates the causal consistency requirement.

To be sequentially consistent, the order in which read data is observed must be the same across all processes. In our example C, D and E observe data in a different order and therefore they are not sequentially consistent.

2.3 Multiparty asynchronous session types

While conversations between distributed processes often require a structured protocol, the programming language primitives to support such communications are typically limited to one-time (send-receive) interactions. More involved sequences of interaction require manual programming labor.

Significant benefits can be expected when programmers are able to specify a conversation as an explicit structure [13]. The aim is to support the realization of more readable, maintainable programs as in general is the case for structured programming. More importantly, the formalization of the conversation facilitates verification of communication properties either statically or at runtime. This may help prevent or at least detect incompatible communications, for example synchronization errors.

Note that the specification of a structured conversation could be processed similar to how other source code is processed by a compiler. After a lexical scan, the specification is parsed into terms. Subsequently these terms are evaluated according to formal rules into a normalized form. Proper evaluation is an indication that the specification is a valid conversation and complies with all properties that are guarded by the formal rules. The rules in essence document acceptable conversation constructs as a mathematical formalism.

Honda et al. propose to use three key elements to structure conversations between processes [13]. The process (a)synchronous interaction must be modelled in *constructs* to make its structure explicit. In addition to providing common features such as send and receive primitives, more complex interactions are supported through *composition operators* that influence the flow of communications. Communication interaction is categorized by *type* to facilitate verification of safety properties and hence ensure compatible communication.

Milner's polyadic π-calculus is used as a foundation for modelling processes and their interactions [22]. This algebra formalizes the syntax and operational semantics of process interactions. Information is exchanged between processes in the form of names which can be used to reference a process or to exchange other data. The calculus defines terms that can be supported via language constructs incorporated as extensions in existing programming languages or specified separately using a domain-specific language such as Scribble [15].

The calculus is enhanced with communication primitives for label branching and delegation [13]. Label branching involves the communication of a label, a value from a defined set of names, to another process. The receiving process uses the label in a conditional expression to branch out to one of its behaviors. Delegation is used by processes to hand-off their role in a conversation to another process. An example use case is a server listener process that receives a request from a client process and delegates the response to this request to a spawned agent process.

The terms in the calculus are annotated with types [24]. Types can be verified to detect and prevent potential runtime failures and deadlocks caused by incompatible communication. For example, communication is incompatible when the sending process presents a numeric value and the receiving process expects a boolean value. In this situation communication can fail despite the fact that the sequence of messages might comply with the required structure.

Structured interactions can involve more than two participating processes. To meet this need, Honda et al. propose to use multiparty asynchronous session types (MPST) [14]. The conversation, a session, relies on asynchronous communication between two or more processes, referred to as parties. Communication between parties may involve multiple channels. Both linearity (causal relationship) and progress (deadlock-free) are guaranteed to be properties of a channel within the context of a single session.

We will use a simple example to show what an MPST conversation might look like. In this example the Buyer, a client process, seeks to purchase a table or a chair from the Seller, a server process. The Seller is an intermediate actor. The articles will be delivered by another

process which is either the Tables or the Chairs process. Figure 2.4 shows the conversa-

```
import Purchase.messages.*;
protocol Purchase {
 role Buyer, Seller ,Tables ,Chairs;

 hello from Buyer to Seller;
 specialOffer from Seller to Buyer;
 choice from Buyer to Seller {
  tableArticle():
   buyTable from Buyer to Seller;
   getTable from Seller to Tables;
   deliveryDate from Tables to Seller;
   deliveryDate from Seller to Buyer;
  chairArticle():
   buyChair from Buyer to Seller;
   getChair from Seller to Chairs;
   deliveryDate from Chairs to Seller;
   deliveryDate from Seller to Buyer;
 }
}
```

Figure 2.4: Global protocol

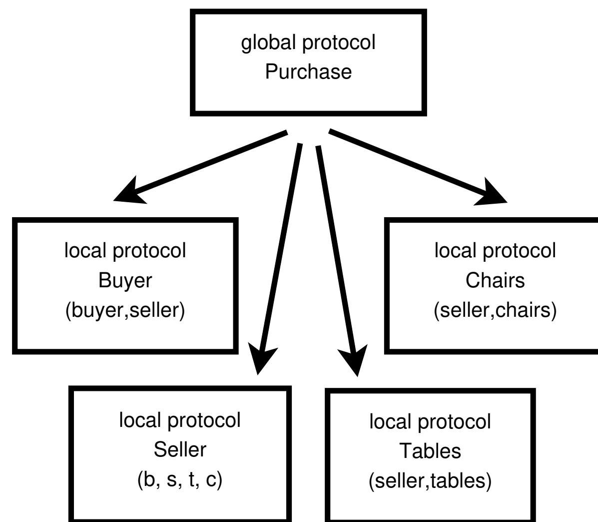


Figure 2.5: Projected local protocols

tion structure for session type Purchase, referred to as the global protocol. Here, the syntax is based on Scribble but any programming language that implements the MPST calculus would do. A role is defined for each participating process. The Buyer sends an initial message to the Seller and then receives a list of special offers. The next message exchange shows how the communication primitive label branching is used. The Buyer sends the Seller a label indicating a choice and the Seller branches to execute the related conversation part of the protocol. In this part, a backoffice conversation starts with another process to get the selected article. Once that part is completed, the Seller reports back the expected delivery date to the Buyer.

Distributed process communications are commonly implemented as one-to-one conversations. Figure 2.5 shows how the *global* protocol Purchase is projected onto multiple *local* protocols that model the conversation from the perspective of one of the parties. For instance the local protocol of party Chairs will only include an interaction between Chairs and Seller. The local protocol of Seller will be similar to the global protocol because it involves interactions with all other roles.

Ultimately, the local protocol code is compiled and linked to the program of the relevant process.

3

PROBLEM ANALYSIS

Data grids aim to maintain coherence between data that they manage. We will now zoom in on a data grid function that requires coherence. Subsequently we discuss an example case where coherence is challenged.

Providing applications with easy access to distributed data is a key function of the data grid as introduced in Section 2.1. This function is implemented through virtualization of underlying storage systems. Access to files is provided via logical structures called data objects. For reasons such as data safety, replicas of a file can exist on storage media distributed across the grid.

In the iRODS implementation of the data grid architecture, the filesystem path of a replica is managed such that it remains coherent with the logical path of the associated data object, only their prefixes differ. Whenever an application requests to update the logical path of a data object, this change is propagated and also applied to the physical paths of all associated replicas.

This coherence feature is used and appreciated by system administrators and a class of high-performance applications. For instance a system administrator can now use regular operating system commands to directly access storage systems and check if any files are infected by a virus. Should this be the case then the path to the replica file can be used to locate the associated data object. Subsequently the data object is annotated with metadata to signal the poor state of the replica. Likewise applications can be authorized to bypass[3] the data grid layers in case they require very high data read access performance. Instead of using data grid functions, they are allowed to directly call storage system I/O functions to access the replica.

The strategy that iRODS follows to implement coherence between data object paths and replica paths is to perform the updates in a fixed order. An iRODS agent process coordinates the update operation. Other data grid processes are contacted in turn to perform the actual

[3]Despite its potential impact on data integrity, the bypass is considered a feature of the system.

updates. First, the agent has the data object path updated. Next, it iterates through an ordered list of replicas and requests the replica path update to be performed. The agent awaits the succesful completion of an update before it initiates the next update request.

Unfortunately the iRODS implementation of coherence between data object path and replica path is subject to a race condition. If two applications concurrently request the data grid to update the path of the same data object, in an odd case the coherence between an data object and its replicas may be lost.

Process A:	dataObj:="n1"			replica:="n1"
Process B:		dataObj:="n2"	replica:="n2"	

Figure 3.1: Potential race condition at data object path update

In Figure 3.1 we demonstrate this race condition in an example where a data object is associated with only one replica. We assume two concurrent requests to update the data object path. Data grid agent process A coordinates the request to set the data object path to `"n1"`. Agent process B coordinates the other request to set the path to `"n2"`. Coherence requires that after applying all the updates both the data object and its replica exhibit an equivalent path. Hence both the data object and its replica should either exhibit path `"n1"` *or* both should have path value `"n2"`. In our example, the interleaved execution of the update operations results in a non-coherent end state: the data object path is `"n2"` whereas the replica path is `"n1"`.

Note that the fixed order strategy as implemented by iRODS is efficient and basic grid configurations may well never experience the above race condition. The race is likely to emerge in configurations with increased chances of request interleaving, for instance when similar grid processes are distributed across computer nodes with different performance characteristics.

Our example showcases a race condition that can be considered archetypical for a non-transactional distributed system. The general case consists of a set of operations on *multiple* items of persistent data, each data item managed by a different process, where the set of operations needs to be executed as if it were a single transaction.

Next to this type of race condition, a distributed system may suffer from other types of race conditions not covered by our research. For example, when a set of operations is executed on a *single* data item, and these operations are executed concurrently, then a race condition can emerge. This type of race condition can be prevented efficiently: We require that all the processes responsible for managing the data item are located on the same node as where the data item itself persists. Now traditional mutual-exclusive locking mechanisms can be used without running the risk of performance degradation from network latency.

4

RESEARCH SCOPE AND METHODS

As demonstrated by our problem analysis, data coherence is a desirable property of a data grid, yet it is not trivial to guarantee this property in current data grid systems. A fixed order strategy as implemented in iRODS does not keep data coherent in all circumstances. Note that even if a solid strategy has been established, the program code for process interactions would still be intertwined with other operations. This scattered implementation is difficult to maintain for software engineers.

Can we improve the management of data coherence in non-transactional distributed systems through a method that supports formal verification of this property? Can we use a domain-specific language such as provided by multiparty asynchronous session types (MPST) for this purpose? These are major challenges that set the context for our research.

Extensive research will be required to answer the above questions thoroughly, even if this research is limited to distributed systems with a data grid architecture. For example here are some aspects that future research might need to address:

- How well can we specify all data grid process interactions using MPST?
- Can MPST be implemented with sufficient performance to meet the needs of data grids?
- Can session types be used or extended to easily detect and prevent all data coherence issues in data grid conversations?
- Are we able to verify and guarantee that data coherence holds as a property for these conversations?
- How can MPST be integrated gently in current data grid architectures? For instance how easy is it for software engineers to model data grid conversations, including data coherence requirements, in a domain-specific language such as Scribble?

4.1 Research questions and methods

Our research contributes a first step to this journey. Applied to a common data grid operation use case, we investigate how data coherence can be modeled in formal temporal logic and how we can express coherence requirements as an extension of a multiparty asynchronous session types based calculus. For our purpose and context, a data grid is a non-transactional[4] distributed system with asynchronous process communications.

Research Question: *How can we express and manage data coherence between a data object and its replicas in a data grid using a domain-specific language based on the multiparty asynchronous session types calculus?*

We limit the scope of data coherence to a value update operation for a single property of the data object, where we require that a related property of all its replicas is updated with the same value.

We address this question via four subquestions. Our methods to address these subquestions can be categorized as design-and-create. For each subquestion we document the scope, our approach, and an overview of deliverables. A list of the software used in our research can be found in Appendix A.

4.2 RQ1: Capture the problem case in a session type model

RQ1 *a) How do we model the problem use case documented in Chapter 3 as a protocol?*
b) How do we establish that this protocol allows for a race condition?

We model the update operation as a protocol that services multiple parallel requests, where each request requires an update on persistent data. Using this approach we learn how well we can express and manage coherence using the existing syntax and semantics of sessions. The requests are scoped to an update operation on a property of a data object that is related to a single replica. We demonstrate how to scale the model to support scenarios with multiple replicas.

We experiment with two different techniques to implement the above protocol. We model this conversation as a global protocol in Scribble [15]. As an alternative approach, we also implement the conversation as a labeled transition system in UPPAAL [2].

Scribble is a natural choice as this description language and compiler have been developed for representing the interaction protocols, in step with the multiparty asynchronous session types theory [14]. Using Scribble, we should be able to reproduce the data race in this protocol by creating a Java test application for each role involved. Scaling the model from one replica to multiple replicas is possible by adding an extra protocol role per replica. We use the Scribble language construct *par* to model interleaved update requests. Unfortunately it turns out that support for interleaved operations, an element key to our model,

[4] Data grids depend on storage systems that do not always provide transaction features such as commit/rollback.

is not yet supported by the Scribble compiler.

Using UPPAAL, we are able to simulate the update operation and reproduce the data race condition. In addition, it allows us to prototype solutions that protect coherence. In order to capture coherence aspects, we use its ability to model asynchronous operations and concrete data transfers. Yet we are unable to use this tool to produce actual Java applications for each role.

Our experiment shows that to protect coherence, we will require a domain specific interaction language that supports three key features: 1) interleaved operations, 2) asynchronous communications, and 3) concrete data transfer operations.

The deliverables are a model of the problem use case documented as a concrete protocol (Scribble) and an implementation of the concrete protocol as a labeled transition system (UPPAAL). Chapter 5 covers the results related to this research question.

4.3 RQ2: Design coherence support as MPST calculus extension

RQ2 *How do we represent coherence as a property of related data?*

We design support for expressing coherence in a new theory of multiparty asynchronous session types. This design allows us to check the invariant relation that we seek to establish between a property of a data object and the related property of a replica. We consider a simple coherence relation where the value of a replica property must remain equivalent to the value of the related data object property.

First, we design a calculus that can meet the requirements that surfaced in RQ1: interleaved operations, asynchronous communications, and concrete data transfers. Interleaved operations are needed to represent concurrent service requests to the system. Asynchronous communications mimic actual communication patterns as commonly found in distributed systems such as data grids. Information from concrete data transfers is needed, to assess if a data transfer will make or break the invariant coherence relation.

In line with theory developments, we base our calculus on a recent formalization of the multiparty asynchronous session types [16]. The calculus can be used to specify a global protocol for interactions. We design the implications for participating roles as endpoint[5] projections. The resulting *local* protocols comply with the grammar of a companion calculus. The deliverable is a mathematical design, documented in Chapter 6.

Next, we extend our theory with support for coherence. We define coherence in terms of temporal logics with respect to a Kripke structure [17]. The coherence model describes a relation between two attributes. We integrate this model as a layer of structural operational semantics in the previously designed calculus.

[5]The term endpoint is used to refer to a process that takes part in an interaction.

The deliverable, again a mathematical design, is discussed in Chapter 7.

4.4 RQ3: Validate the coherence-extended calculus

RQ3 *a) How can we establish that the projection of global protocols to local protocols, as designed for our calculus, is sound and complete?*
b) How can we establish that the expressiveness of our calculus is sufficient to support verification of data coherence for concrete interactions?

Both research questions are answered using model checking. To prepare for model checking, we implement the calculus, participating roles and communication channels along with a concrete interaction protocol as a labeled transition system (LTS). The concrete protocol will trigger actions that are processed by the remainder of the system in line with the operational semantics of the calculus.

We select mCRL2 as a model checking toolset suitable for our purpose [5]. mCRL2 is able to both model check a property of an LTS (required to answer RQ3b), as well as to establish if two LTS systems are sufficiently equivalent to be considered bisimilar (required to answer RQ3a). In addition, the mCRL2 modeling language syntax matches with a large part of our calculus syntax, which saves us implementation time. Other tools that we have considered include UPPAAL (optionally with TIGA extension) and LTSmin [2][4]. UPPAAL lacks support for bisimulation checking and provides limited support for checking our coherence temporal logic property. The installation and configuration of LTSmin is more involved than mCRL2, while we are unlikely to require many of its features.

To answer RQ3a, we use samples of concrete global protocols to create instances of LTS. One sample, named `good`, is a protocol that can be realized by endpoints. Another sample, named `bad`, serves as a counter example. The counter example fails to meet one of the well-formedness conditions identified by Jongmans and Yoshida [16]. Per sample, we create an additional LTS using the projected local protocol. We use mCRL2 to establish that the two LTS instances related to the `good` sample are *weakly bisimilar*. Weakly bisimilar systems behave in an equivalent way and transition in sync (disregarding internal actions). The bisimulation result validates that our projection is sound and complete.

To answer RQ3b, we transform the coherence property defined in Section 7.3 to a format[6] that is accepted by mCRL2. We use this formula to verify samples of global protocols configured in the above mentioned LTS. The results are compared with expected values.

The deliverable is a verified model, discussed in Chapter 8.

[6] mCRL2 accepts model checking formula specified in μ-calculus

4.5 RQ4: Prototype language extensions that protect coherence

RQ4 *Which additional domain-specific language constructs are needed to express and provide support for coherence of related data for the use case in RQ1?*

We aim to find out if we can facilitate software engineers to conveniently specify coherence requirements as part of a session protocol. Is this a straight-forward implementation of the session calculus in a domain-specific language, or will it demand more complex compositions on top of the calculus, such as macro-like features? For practical reasons, we limit the implementation of constructs to those needed to support our RQ1 use case.

The protocol that represents our problem use case is our initial test case. We use the calculus implementation in mCRL2 as created in RQ3. We verify that the coherence property does not hold for this protocol. Next, we enhance the session protocol of our use case to express that the data object and a single replica should remain coherent, using a prototype extension of the calculus. Subsequently, we establish that the new session protocol guarantees coherence as a property of the related data. Further, we establish that this guarantee holds when we scale the use case to multiple replicas. Our prototype supports sequential consistency, one of the consistency models discussed in Section 2.2.

The deliverable is a prototype and a discussion of strategies that can be used to protect coherence. Chapter 9 covers the results related to this research question.

4.6 Research validation

We use the seven criteria provided by Vaandrager to reflect on the quality of our use case model [29]. A good model has a clearly specified object of modelling, a clearly specified purpose, is traceable, is truthful, is simple, is extensible and reusable, and is designed and encoded for interoperability and sharing of semantics. Chapter 5 includes this reflection.

We validate the designed calculus as well as the coherence property formula by model checking an implementation of the calculus. More details can be found in Chapter 8.

5

Model the problem case as a protocol

The informal problem analysis in Chapter 3 discusses a use case where an iRODS data object is moved. The move operation requires an update of the logical path attribute of the data object as well as an update of the related path of each replica associated with this data object. The interleaved processing of multiple move requests for the same data object can result in a data race. While we capture this interaction in a formal specification using two different techniques, we learn about requirements for coherence support.

5.1 Problem case modeled as protocol in Scribble

We model the interaction between processes involved in our use case as a Scribble protocol [15]. Figure 5.1 shows the resulting specification. The sequence of interactions that take place to process a move are specified in the global subprotocol `MoveObject`. The data object is referenced as `ObjName`. A replica is referenced by its managing process and the relation with a data object.

In our model, we will assume that each replica is managed by a separate process. In reality, our data grid uses a combination of processes and threads to respond to replica-related requests. We argue that our simplification is justified, as we aim to model the persistent state of the replica, rather than the responding thread. Ultimately, all changes applied to a replica will be executed in sequence.

The updated property, a logical or physical path, is depicted as `CollName`. Role `A` represents a data grid agent process, whereas roles `D` and `R1` depict processes that manage respectively the data object and the replica. The data grid agent coordinates the processing of a move request received from a client application.

First, the agent sends a message `MoveDataObject` to role `D` to have the data object itself updated. It waits until `D` acknowledges that this action has completed.

Next, the agent orders associated replicas to be updated, each replica in turn, using a fixed

```
// Global Protocol for performing interleaved Data grid updates
//
// A = agent  D = data object manager   R = replica manager
// option: To support multiple replicas, add more roles besides R1

module move;
type <java> "java.lang.String" from "rt.jar" as CollName;
type <java> "java.lang.String" from "rt.jar" as ObjName;

global protocol Move(role A1, role A2, role D, role R1) {
    par {
        do MoveObject(A1, D, R1);
        } and {
        do MoveObject(A2, D, R1);
    }
}

global protocol MoveObject(role A, role D, role R1) {
    MoveDataObject(ObjName,TargetCollName) from A to D;
    Ack() from D to A;
    MoveReplica(ObjName,TargetCollName) from A to R1;
    Ack() from R1 to A;
}
```

Figure 5.1: Scribble protocol for data grid update

order. A data object has at least a single replica, which is the configuration used in our specification. Role `R1` manages this replica. The protocol specification can scale to support a data object with more replicas by merely adding replica roles and `MoveReplica`/`Ack` messages.

Interleaved processing of moves is specified in global protocol `Move`, where roles `A1` and `A2` represent two agent processes that concurrently process a move. When roles `A1` and `A2` send messages with the same concrete value for `ObjName` and differing values for `CollName` then a data race, as described in our problem case, can occur. The interleaving factor of the model can be scaled by adding more agents.

Unfortunately, compilation of the `Move` protocol leads to endpoint Java code that throws a nullpointer exception at runtime. Investigation reveals that this exception occurs whenever we specify interleaved interactions in conjunction with Scribble compiler release 0.4.3.

While we have used a Scribble syntax for parallel interactions, this syntax is not completely and consistently supported by the Scribble compiler. The original design for the Scribble language specified an operator `&` to be used with unordered (parallel) interactions [15]. Scribble compiler release 0.4.3 accepts the syntax `par` as we have used for our use case protocol, yet the generated endpoint Java code is ineffective with respect to this operation. Support for parallel interaction syntax, a key requirement for our purpose, has been dropped in a subsequent version of the compiler.

5.2 Problem case modeled as LTS in UPPAAL

As protocol compilation using the Scribble compiler has not proven fruitful, we resort to simulation techniques for testing our protocol. To this end, we model the protocol as a labeled transition system (LTS) in UPPAAL [2].

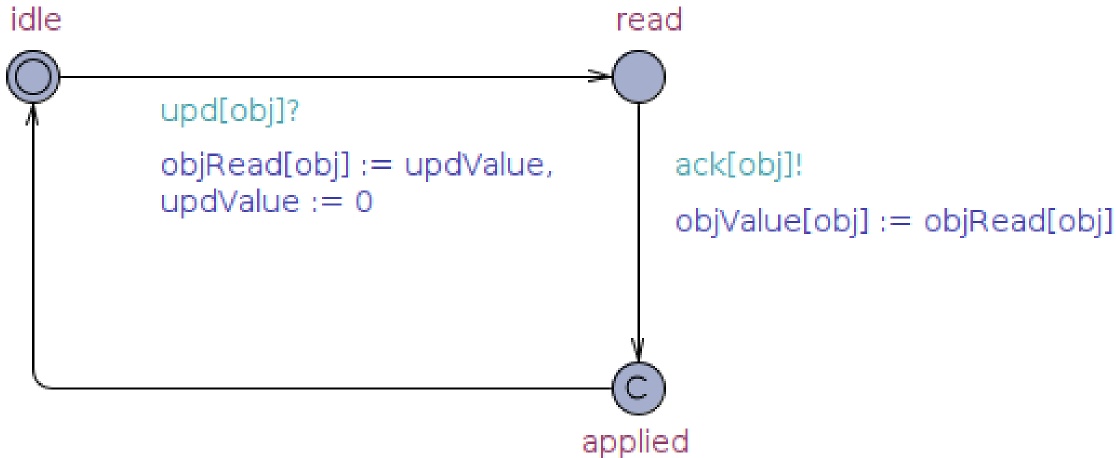


Figure 5.2: UPPAAL template for an Object (Data Object or Replica)

We use a template named `Object`, as depicted in Figure 5.2, to represent roles that manage either the data object or a replica. The instance `Object`[$obj \mapsto 0$] manages the data object. Other `Object` instances ($obj > 0$) manage a replica. Persistent state of the object is captured in global variables. The source code containing global declarations is listed in Appendix B.

Object roles receive values from agents and update their persistent variable `objValue` with this new value. Once the value is stored, an acknowledge message is sent back to the agent. We use multiple state transitions to reflect the impact of asynchronous communication on the state of the managed variable. The `upd` channel action synchronizes with a message sent by an agent. The receipt of this message (implicitly) takes place at the next state transition.

Agent roles are represented as instances of the `AgentNaive` template, shown in Figure 5.3. An agent iteratively selects an object role, sends it an update request and waits until it receives an acknowledgement message back.

Model checking confirms that a system with multiple naive agents is subject to a data race condition. We note that our simulation has required the use and inspection of concrete data transfers in combination with modeling asynchronous communications.

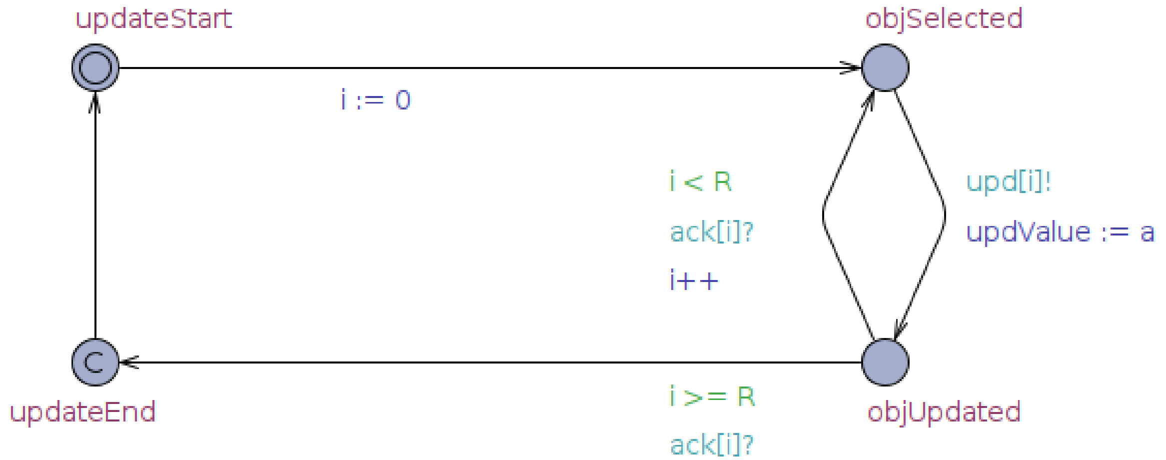


Figure 5.3: UPPAAL template for AgentNaive

5.3 Prototyping data race avoidance

Using UPPAAL, we can prototype ideas to resolve the data race as seen in our use case. We assume that a strategy will involve some contract between the processes involved, to enforce that updates take place in an acceptable order. We aim to minimize the impact of exclusive access mechanisms on throughput performance.

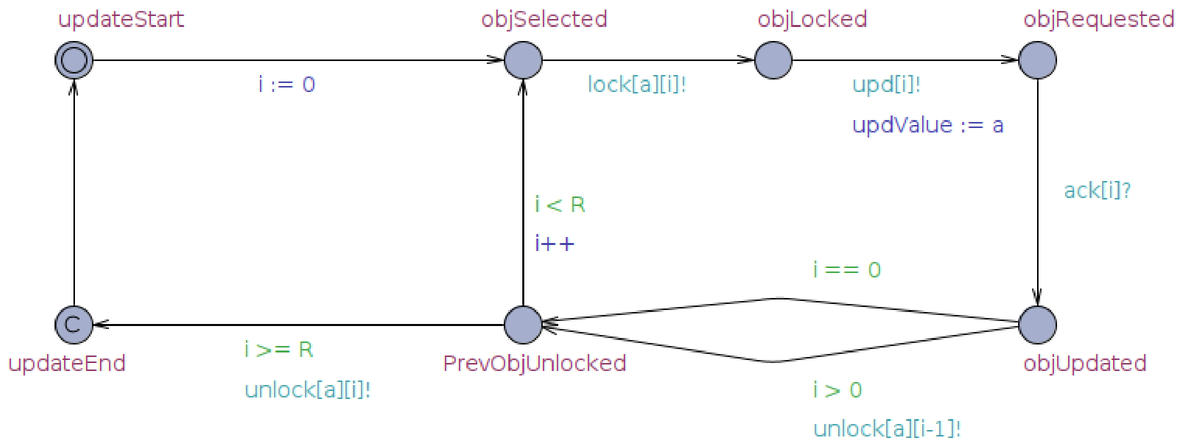


Figure 5.4: UPPAAL template for AgentMilestone

Figure 5.4 shows an agent that uses a *milestone* approach to prevent a data race. Like our naive agent, the milestone agent iteratively requests objects (data/replica) to be updated. However, it deploys a locking mechanism to enforce exclusive access to an object. A lock is acquired[7] just prior to an update request for the object being locked. A lock is released only *after* a lock for the subsequent object is obtained, which is considered the next milestone. Note that this milestone locking mechanism relies on a contract between agents, where

[7]The agent synchronizes with a `Lock` process associated with a particular data object or replica. The `Lock` process, not shown here, merely oscillates between the states `locked` and `unlocked`.

all agents use the same fixed iteration order. Model checking confirms that the data race condition has disappeared after we replace the naive agents by milestone agents.

The milestone approach may reduce the impact of locking on throughput, for scenarios where a data object is associated with more than one replica. For instance, when an agent is updating the third replica of a data object, then another agent may simultaneously update the first replica of the same set.

The *chained-coherence* strategies, discussed in Chapter 9, have been inspired by the results gained from the above milestone agent experiment.

5.4 Validation of problem case model

We reflect on the quality of our Scribble model and our UPPAAL model using criteria listed by Vaandrager [29]. A good model has a clearly specified object of modelling, a clearly specified purpose, is traceable, is truthful, is simple, is extensible and reusable, and is designed and encoded for interoperability and sharing of semantics.

We have a clear *object of modelling* as outlined in Chapter 3. We model an interaction between data grid processes, that intends to update a data object property and a similar property at related replicas in sync.

The model's *purpose* is to serve as a testcase for a domain-specific interaction language. The language should allow us to reason about coherence.

All elements of the Scribble model and the UPPAAL can be *traced* back to elements of the real-world object being modelled. Traceability of the UPPAAL model is less transparant compared to the Scribble model. For instance, the asynchronous nature of the update message is implemented as a synchronization followed by another state transition.

The UPPAAL model is also less *thruthful* than the Scribble model. In particular the acknowledgement message is exchanged via synchronous communication while actual data grid processes communicate using asynchronous communications. We argue that for our purpose this is an acceptable limitation as the state of the persistent property will not be affected by this message exchange.

The Scribble model has sufficient predictive power while remaining *simple*. In the UPPAAL model however, we have added extra states that do not relate to any aspect of the modeled object. The added states are `updateEnd` in the agent templates and `applied` in the object template. They have been added solely to ease model checking. We eliminate the impact of the extra states on the state space by marking them `committed`[8].

The *extensibility and reusability* of our Scribble model is limited. Scaling the model to include more agents or replicas is possible, yet it requires manual changes to the code. For instance, adding a replica requires changes to the list of roles in the `protocol` statements

[8] In UPPAAL, the transition to a committed state is executed atomically with a transition to the next state.

and the addition of an update interaction statement. In contrast, the UPPAAL model is highly extensible and reusable as a result of using templates to describe role types. Adding another agent or replica is accomplished by changing a configuration parameter. We reuse the `Object` template in our milestone agent prototype.

As our models are based on existing tools and grammars, they can in theory *interoperate* with other specifications. For instance, our Scribble protocol could be embedded as a sub-protocol in another specification.

6

An MPST based calculus to support asynchronous data transfers

In this chapter, we design a calculus that serves to capture the interactions between processes in a separate program (protocol) specification. In a next chapter, we will build on this calculus and extend it with features that allow us to detect and ideally prevent loss of coherence.

Our calculus takes a system perspective, while each process will only execute a part of the interactions. We include support for the process perspective through a companion calculus. A projection operator links the calculi.

6.1 A calculus for concrete asynchronous interactions

Our interaction calculus is based on an MPST calculus by Jongmans and Yoshida, which we adapt to support asynchronous communications of concrete data transfer messages [16].

Let $\mathbb{R}$ denote the set of roles, representing processes that potentially participate in the global protocol.
Let $tup(R)$ denote a set of role tuples, representing combinations of two interacting processes, generated by $tup(R) = \{(p,q) \mid p \neq q \land p,q \in R \subset \mathbb{R}\}$.
Let $\mathbb{Q}$ denote the set of all sequences of data, ranged over by Q, representing queued content in channels.
Let $\{X_1, X_2, \ldots\}$ denote a set of recursion variables.
Let $\mathbb{G}$ be a language, defined as a set of protocols ranged over by G (the grammar for $\mathbb{G}$ is defined below).
Let R_G denote the set of roles involved in G where $R_G \subset \mathbb{R}$.
Let $C : tup(R_G) \to \mathbb{Q}$ denote the set of unidirectional communication channels between roles involved in G, with their content. We use the notation pq as the name for the channel from role p to role q. Its content, a queue of data, is referenced by $C(pq)$.
Let $c_{pq}.v$ be a transfer of data over channel pq where v denotes a value that is transferred

from role p to role q. A transfer $c_{pq}.v$ is performed using a send followed by a receive action, denoted respectively as $c_{pq}!v$ and $c_{pq}?v$.

$\mathbb{G}$ is generated by the following grammar:

$G ::= 1 \mid F_{pq}.v \mid c_{pq}.v \mid G_1 + G_2 \mid G_1 \cdot G_2 \mid G_1 \| G_2 \mid X \mid \langle X_k | \{X_i \mapsto G_i\}_{i \in I} \rangle \;\; (k \in I)$

1 denotes a *skip*. This grammar construct is used to structure operational semantics rules. Rules that define valid transitions are separated from rules that define termination. Skip is not intended to be used explicitly in instances of the language.

$F_{pq}.v$ denotes a *full*. Like skip, this construct must not be used by programmers in instances of the language. It is used in transition rules to allow receive actions only after the channel is full of content from a corresponding send action.

X denotes a recursion variable. The operators $+$ and $\cdot$ and $\|$ represent respectively alternative, sequential and interleaved compositions. The interleaved compositions are parallel compositions without interaction between the operands (free merge) [3].

The notation $\langle X_k \mid E \rangle$ reads as: *the term specified as variable X_k is substituted by another term using the provided specification E* [10]. In our grammar, E is a recursive specification that maps X_i to a corresponding term G_i for a finite set of i. Recursion is facilitated by the combined presence of the recursion variable X and $\langle X \mid E \rangle$ as terms in G. For instance, starting at a term X_k, this term is listed in the recursive specification and maps to term G_k. Since X is listed as an option in the definition of term, G_k may take the form of a recursion variable. Assuming that G_k indeed takes the form of X_j, the new term meets the recursive specification. A next level of recursion is reached as we substitute X_j by its corresponding term G_j.

$$\frac{}{c_{pq}.v \xrightarrow{c_{pq}!v} F_{pq}.v} \qquad \frac{}{F_{pq}.v \xrightarrow{c_{pq}?v} 1} \qquad \frac{G_1 \xrightarrow{\alpha} G_1'}{G_1 \cdot G_2 \xrightarrow{\alpha} G_1' \cdot G_2} \qquad \frac{G_1 \downarrow \quad G_2 \xrightarrow{\alpha} G_2'}{G_1 \cdot G_2 \xrightarrow{\alpha} G_2'}$$

$$\frac{G_1 \xrightarrow{\alpha} G_1'}{G_1 + G_2 \xrightarrow{\alpha} G_1'} \qquad \frac{G_2 \xrightarrow{\alpha} G_2'}{G_1 + G_2 \xrightarrow{\alpha} G_2'} \qquad \frac{G_1 \xrightarrow{\alpha} G_1'}{G_1 \| G_2 \xrightarrow{\alpha} G_1' \| G_2} \qquad \frac{G_2 \xrightarrow{\alpha} G_2'}{G_1 \| G_2 \xrightarrow{\alpha} G_1 \| G_2'}$$

$$\frac{sub(E, E(X)) \xrightarrow{\alpha} G'}{\langle X | E \rangle \xrightarrow{\alpha} G'}$$

$$\frac{}{1 \downarrow} \qquad \frac{G_1 \downarrow}{G_1 + G_2 \downarrow} \qquad \frac{G_2 \downarrow}{G_1 + G_2 \downarrow} \qquad \frac{G_1 \downarrow \quad G_2 \downarrow}{G_1 \cdot G_2 \downarrow} \qquad \frac{G_1 \downarrow \quad G_2 \downarrow}{G_1 \| G_2 \downarrow} \qquad \frac{sub(E, E(X)) \downarrow}{\langle X \mid E \rangle \downarrow}$$

Figure 6.1: Structural operational semantics for the global language

Figure 6.1 specifies the structural operational semantics for the above grammar. It includes rules for transition and termination. We use the notation $G \xrightarrow{\alpha} G'$ to denote a transition

from G to G' by executing an atomic operation α. This atomic operation α is defined as:
$\alpha ::= c_{pq}!v \mid c_{pq}?v \quad (p \neq q)$
The specification $G\downarrow$ denotes a successful termination.
A helper function $sub(E,G)$ allows us to retain a notion of the applied recursive specification. Note that this specification is consumed in the *substituted* form of a recursive term. Our helper function is defined as:
Let $\mathbb{X} \rightharpoonup \mathbb{G}$ be the set of partial functions that define recursive specifications, ranged over by E. Let $sub(E,G)$ denote the simultaneous substitution of term $E(X)$ for each recursion variable X in G.

Informally, the transition rules state that a data transfer is commenced by processing a send action. The data transfer is completed by processing a receive action on the same channel. Sequential compositions are reduced by first reducing the prefix term and then reducing the suffix term after the prefix has terminated. Alternative compositions are reduced by reducing one of its component terms. The interleaved composition is reduced by reducing both component terms. A recursive composition term is reduced by reducing a version of the term where all recursion variables have been substituted. The termination rules state that a skip can always terminate. Alternative compositions terminate when either one of their terms terminate. Sequential and interleaved compositions terminate when both terms terminate. A recursive composition terminates when its version with substituted terms terminates.

We restrict our resulting global language to protocols that do not include any occurences of a *full* or *skip*, nor include any *free* recursion variables. In addition, we require that there will always be only a single option for processing an action. These preconditions are referred to as full-free, 1-free, closed and deterministic [16].

6.2 Endpoint perspective: calculus for local language

The interaction language takes a system view, it is a *global* language. The global language is less suited to describe actions from the perspective of an individual role. For instance, while two roles might be engaged in communication, a third role might be required to wait. Such an idle action at an endpoint is not part of the global language.

We continue by defining a calculus for the *local* language related to our interactions. Our local language takes the perspective of a single role (process) that participates in a protocol expressed in the global interaction language.

6.2.1 Local language for a single endpoint

The notational conventions of the global grammar apply to the local grammar as well.
We define the language of local protocols as the set $\mathbb{L}$, ranged over by L.
Let R_L be the set of role names $\{r_1, r_2, \ldots\}$ involved in L where $R_L \subset \mathbb{R}$.
Let $C : tup(R_L) \rightarrow \mathbb{Q}$ denote the set of unidirectional communication channels between

roles involved in L, with their content.

$\mathbb{L}$ is generated by the following grammar:

$\alpha ::= \tau \mid c_{pq}!v \mid c_{pq}?v \;\; (p \neq q)$

$L ::= 1 \mid \alpha \mid L_1 + L_2 \mid L_1 \cdot L_2 \mid L_1 \| L_2 \mid X \mid \langle X_k \mid \{X_i \mapsto L_i\}_{i \in I}\rangle \;\; (k \in I)$

The element α denotes an atomic action. An idle action is denoted by τ.

$$\frac{}{c_{pq}!v \xrightarrow{c_{pq}!v} 1} \qquad \frac{}{c_{pq}?v \xrightarrow{c_{pq}?v} 1} \qquad \frac{}{\tau \xrightarrow{\tau} 1}$$

$$\frac{L_1 \xrightarrow{\alpha} L_1'}{L_1 \cdot L_2 \xrightarrow{\alpha} L_1' \cdot L_2} \qquad \frac{L_1 \downarrow \quad L_2 \xrightarrow{\alpha} L_2'}{L_1 \cdot L_2 \xrightarrow{\alpha} L_2'} \qquad \frac{L_1 \xrightarrow{\alpha} L_1'}{L_1 + L_2 \xrightarrow{\alpha} L_1'} \qquad \frac{L_2 \xrightarrow{\alpha} L_2'}{L_1 + L_2 \xrightarrow{\alpha} L_2'}$$

$$\frac{L_1 \xrightarrow{\alpha} L_1'}{L_1 \| L_2 \xrightarrow{\alpha} L_1' \| L_2} \qquad \frac{L_2 \xrightarrow{\alpha} L_2'}{L_1 \| L_2 \xrightarrow{\alpha} L_1 \| L_2'} \qquad \frac{sub(E, E(X)) \xrightarrow{\alpha} L'}{\langle X|E\rangle \xrightarrow{\alpha} L'}$$

$$\frac{}{1\downarrow} \qquad \frac{L_1\downarrow}{L_1 + L_2\downarrow} \qquad \frac{L_2\downarrow}{L_1 + L_2\downarrow} \qquad \frac{L_1\downarrow \quad L_2\downarrow}{L_1 \cdot L_2\downarrow} \qquad \frac{L_1\downarrow \quad L_2\downarrow}{L_1 \| L_2\downarrow} \qquad \frac{sub(E, E(X))\downarrow}{\langle X \mid E\rangle\downarrow}$$

Figure 6.2: Local language structural operational semantics

The operational semantics of our local language are listed in Figure 6.2. A transition $L \xrightarrow{\alpha} L'$ is performed by executing the atomic action α. Informally, the transition rules specify that a send, receive or τ is reduced by executing the related action. Sequential compositions are reduced by first reducing the prefix term and then reducing the suffix term after the prefix has terminated. Alternative compositions are reduced by reducing one of its component terms. The interleaved composition is reduced by reducing both component terms. A recursive composition term is reduced by reducing a version of the term where all recursion variables have been substituted. The termination rules state that a skip can always terminate. Alternative compositions terminate when either one of their terms terminate. Sequential and interleaved compositions terminate when both terms terminate. A recursive composition terminates when its version with substituted terms terminates.

The preconditions that restrict our local language are similar to conditions stated for the global language: we only consider local protocols that are 1-free, closed and deterministic.

6.2.2 Local language for a group of endpoints

A group of endpoints represents the local perspective of multiple endpoints (distributed processes) as they participate in an interaction. This concept will be useful in Section 6.3 where we relate global interaction languages to local languages.

Let $\mathbb{R} \rightharpoonup \mathbb{L}$ denote the set of all groups of local protocols, ranged over by Z. Hence $Z(r)$ refers to the protocol of the endpoint with role r in this group of endpoints.

$$\frac{Z(r) \xrightarrow{\alpha} L'_r}{Z \xrightarrow{\alpha} Z[r \mapsto L'_r]} \qquad \frac{Z(r)\downarrow \quad \textbf{for all } r \in \text{dom } Z}{Z\downarrow}$$

Figure 6.3: Structural operational semantics extended for group of endpoints

To cater for a group of endpoints, our local language operational semantics are extended as shown in Figure 6.3. Informally, a group is reduced when a local term of one of the roles is reduced through either a send, receive or idle action. The group terminates when all of its members have terminated.

6.2.3 Local language for a system of endpoints and channels

In addition to group semantics, we need to add channel behavior to describe the system from a local perspective. We reuse the global language channel related definitions in our local language.

$$\frac{C(pq) = Q, \quad Z \xrightarrow{c_{pq}!v} Z'}{C, Z \xrightarrow{c_{pq}!v} C[pq \mapsto Q \cdot v], Z'} \qquad \frac{C(pq) = v \cdot Q, \quad Z \xrightarrow{c_{pq}?v} Z'}{C, Z \xrightarrow{c_{pq}?v} C[pq \mapsto Q], Z'} \qquad \frac{Z \xrightarrow{\tau} Z'}{C, Z \xrightarrow{\tau} C, Z'} \qquad \frac{Z\downarrow}{C, Z\downarrow}$$

Figure 6.4: Structural operational semantics, channels layer

The added structural operational semantics layer for channels is shown in Figure 6.4. Informally, a send action will add a data transfer message to the back of the channel queue of the channel used. A receive action removes a data transfer message from the front of the related channel's queue, under the provision that at least one message is queued. An idle action has no impact on the state of channels. The system terminates when all its endpoints terminate.

6.3 Endpoint projection

Behaviors of participating roles and channels must add up to exactly meet an interaction protocol specification in the global language. Therefore we project specifications from the global language to a group of endpoint specifications and related channels.

First, we define a projection operator to map a global protocol[9] G to a group of local protocols combined with channels. The projection is correct if the global protocol G and the composition of the group of local protocols L with channels are operationally equivalent, provided that the global protocol can be realized by the endpoints [16].

[9]Note that channels are a function of the global protocol, their projection is implicit.

Informally, the projection of a data transfer depends on the role: it is projected to a send if the role is the source of the data transfer, to a receive if the role is the destination of the data transfer, and to an idle action if the role is not involved in the data transfer. A full is projected to a receive when the role is listed as its destination, and projected to an idle operation in all other cases. Projections of all other forms of global terms on a role are homomorphic, they result in a similar local term. The projection of a global protocol onto a set of roles is the corresponding group of projections, provided that the set of roles is not empty and there exists a local protocol for at least each role in the set.

Our projection operator is defined as:

$$c_{pq}.v \upharpoonright r = \begin{cases} c_{rq}!v & \text{if } r = p \wedge r \neq q \\ c_{pr}?v & \text{if } r \neq p \wedge r = q \\ \tau & \text{if } r \neq p \wedge r \neq q \end{cases}$$

$$F_{pq}.v \upharpoonright r = \begin{cases} c_{pr}?v & \text{if } r = q \\ \tau & \text{if } r \neq q \end{cases}$$

$$\begin{aligned} G \upharpoonright r &= G && \text{if } G \in \{1\} \cup \mathbb{X} \\ (G_1 * G_2) \upharpoonright r &= (G_1 \upharpoonright r) * (G_2 \upharpoonright r) && \text{if } * \in \{\cdot + ||\} \\ \langle X \mid E \rangle \upharpoonright r &= \langle X \mid E \upharpoonright r \rangle \\ E \upharpoonright r &= \{X \mapsto E(X) \upharpoonright r \mid X \in dom(E)\} \\ G \upharpoonright\upharpoonright R_G &= \{r \mapsto G \upharpoonright r \mid r \in R_L\} && \text{if } R_G \subseteq R_L \neq \emptyset \end{aligned}$$

Is this projection sound and complete? We will validate our projection later[10], after we have extended our calculus with features that allow us to reason about coherence.

[10]The validation of our projection is discussed in Section 8.2.

7

Coherence model and integration in calculus

In this chapter we extend our theory to support reasoning about coherence. First, we need to define more precisely what we mean by *coherence*. As it turns out, coherence requires that the order of receive operations meets certains criteria. We model these criteria as a property in temporal logic. Subsequently, to be able to model-check this property, we incorporate information in the operational semantics of our calculus.

7.1 Exploring coherence

We will refer to attributes as *process attributes* when their value is influenced only by messages received through process interactions. Informally, we consider a set of process attributes to be *coherent* if and only if there exists an invariant equivalence relationship between their values.

Our context involves attributes of distributed processes. A requirement to atomically update all related attributes can only be fullfilled at the cost of significant performance degradation. To avoid such a requirement, we will allow for an attribute change that may *temporarily* break the equivalence relationship, provided that until the equivalence relation is restored for the entire set of related attributes: 1) the attribute may not change again and 2) eventually all other attributes in the set apply an equivalent change and 3) all the other attributes do no apply a change that is not equivalent.

This slight relaxation of the invariant supports sequential consistency across coherent attributes. Since the updates themselves are ordered, all processes view attribute changes in the same sequential order.

To establish a formal definition of coherence, first we explore coherence for a set consisting of two attributes. Later we will broaden our scope to an arbitrary number of attributes. We model a composition of processes P and Q. A state will relate to a combination of 1) the

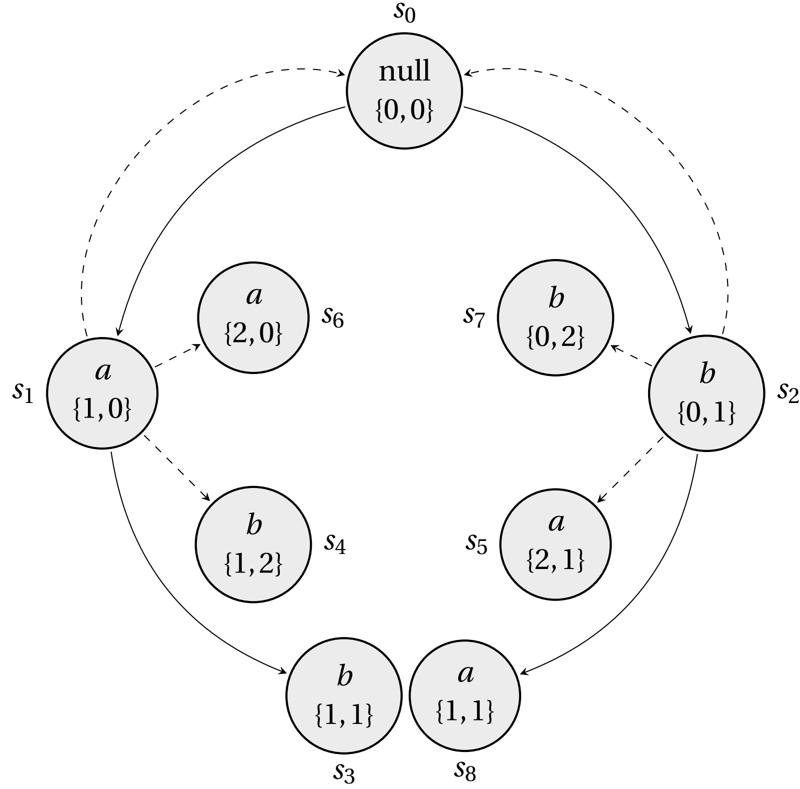


Figure 7.1: Example execution paths for M, incoherent paths are dashed

global session state which includes a composition of P and Q, 2) the last attribute that was changed and 3) the current values of the attributes a (owned by P) and b (owned by Q).

Figure 7.1 shows execution paths at an example starting state s_0 where $[action \mapsto null, a \mapsto 0, b \mapsto 0]$. Note that only the coherence related aspects of the state are shown. Once one of the attribute values has changed, shown in states s_1 and s_2, the other attribute must change to the *same* new value in the next state to safeguard coherence between the attributes. States s_3 and s_8 meet this condition while any other execution path (dashed) leads to loss of coherence.

7.2 Coherence modeled as state transition graph

We define coherence with respect to a Kripke structure $M =< S, R, L >$ [17]. Using the inductive definition of CTL temporal logics by Clarke et al. [8], we document our model semantics as follows:
S is the set of states, $R \subseteq S \times S$ is the transition relation, $L : S \to 2^{AP}$ is a labeling function that maps states to atomic propositions AP. If $p \in AP$ then p is a state formula.
Let A denote the set of all attribute names.
Let D denote the set of all data.
State $s \in S$ is a triple (a, m, G) such that $a \in A$ and $m : A \to D$ is a function from A to D and G denotes the state of our global interaction protocol.

Let π denote a path in M as an infinite sequence of states $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$.
Let π^i denote the suffix of a path starting at state s_i.
Let f, f_1, f_2 denote state formulas and g, g_1, g_2 denote path formulas.

$M, s \models$	p	iff $p \in L(s)$
$M, s \models$	$\neg f$	iff $M, s \not\models f$
$M, s \models$	$f_1 \wedge f_2$	iff $s \models f_1$ and $s \models f_2$
$M, s \models$	$f_1 \vee f_2$	iff $s \models f_1$ or $s \models f_2$
$M, s \models$	$f_1 \implies f_2$	iff $M, s \models f_1$ implies $M, s \models f_2$
$M, (a, m, G) \models$	$last(b)$	iff $a = b$
$M, (a, m, G) \models$	$b = c$	iff $m(b) = m(c)$
$M, \pi \models$	$\neg g$	iff $M, \pi \not\models g$
$M, \pi \models$	$g_1 \wedge g_2$	iff $M, \pi \models g_1$ and $M, \pi \models g_2$
$M, \pi \models$	$g_1 \vee g_2$	iff $M, \pi \models g_1$ or $M, \pi \models g_2$
$M, \pi \models$	$\mathrm{X}(g)$	iff $M, \pi^1 \models g$
$M, \pi \models$	$\mathrm{G}(g)$	iff for all $i \geq 0$, $M, \pi^i \models g$
$M, \pi \models$	$g_1 \mathrm{U} g_2$	iff there exists a $k \geq 0$ such that $M, \pi^k \models g_2$ and for all $0 \leq i < k$, $M, \pi^i \models g_1$
$M, s \models$	$\mathrm{A}(g)$	iff for every path π starting from s is $M, \pi \models g$

7.3 Definition of the coherence property

Using the above model, coherence is inductively defined as a symmetric relationship between an arbitrary set of process attributes. We use $AG(g)$ as a shorthand for $A(G(g))$ and $AX(g)$ for $A(X(g))$.

Definition 7.3.1. The property "attribute b follows attribute a" is:

$$follows(a, b) = \mathrm{AG}(\quad last(a) \wedge a \neq b \implies \mathrm{AX}(\mathrm{A}(\neg last(a) \wedge \neg last(b)\ \mathrm{U}\ last(b) \wedge a = b)\ )$$

Attribute b follows attribute a iff after the value of a is changed, attribute a and b remain unchanged until the same change is applied to b.

Definition 7.3.2. "The property "attribute a and b are coherent" is:

$$coherent(a, b) = follows(a, b) \wedge follows(b, a)$$

Coherence is a symmetric relationship between attributes, such that whenever the value of either one of the attributes is changed, the related attribute must follow suit.

Definition 7.3.3. Let C denote a set of attributes. The property "attribute set C is coherent" is defined as:

$$coherent(C) = \bigwedge\{coherent(a, b) \mid a, b \in C\}$$

The set is coherent only if all possible combinations of its members are coherent. This implies that all members apply the requested attribute value update before they consider any new update requests. For instance, suppose that all attributes of a set $a_1 \ldots a_n$ initially have value 0 and we apply the update $[a_2 \mapsto 1]$. The relation $coherent(a_1, a_2)$ requires that

a_1 must update to the *same* value as a_2, before both a_1 and a_2 are allowed to apply any other changes. The same restriction applies to the other attributes. An update $[a_3 \mapsto 2]$ would break the relation $\mathtt{coherent}(a_2, a_3)$. Since $a_2 \neq a_3$ and $a_2 = 1$, the only acceptable change for a_3 in this state is $[a_3 \mapsto 1]$.

7.4 Calculus extended with coherence model

The designed calculus does not yet track the impact of state transitions on coherence. We will now add the coherence model to this calculus.

7.4.1 Coherence model as global language extension

Figure 7.2 shows the integration of the coherence model, defined in Section 7.2, as an extension of the global language using an extra layer of operational semantics. Our operational semantics assume that roles manage at most a *single* attribute. Conveniently, the name of an attribute is derived from the corresponding role name. This suffices to demonstrate how coherence can be expressed. More attributes per role could be supported by adding a target attribute name to send and receive actions alongside the transferred value.

$$\frac{G \xrightarrow{c_{pq}!v} G'}{(a, m, G) \xrightarrow{c_{pq}!v} (a, m, G')} \qquad \frac{G \xrightarrow{c_{pq}?v} G'}{(a, m, G) \xrightarrow{c_{pq}?v} (q, m[q \mapsto v], G')} \qquad \frac{G \downarrow}{(a, m, G) \downarrow}$$

Figure 7.2: Additional layer of global language operational semantics to integrate coherence model

Informally, a send action does not change any attributes. A receive action updates the targeted attribute with a new value and registers that this attribute has been most recently changed. The system extended with the model terminates when the system terminates.

7.4.2 Coherence as local language extension

We annotate the local language system perspective with the coherence model. This produces an additional layer in the rules of the operational semantics similar to the extension of the global language, except that a rule for idle actions is introduced.

The operational semantics layer is displayed in Figure 7.3. Informally, send or idle actions do not change any coherence related attributes. A receive action updates the value of an attribute and registers this attribute as the most recently changed attribute. The system with coherence annotations terminates when the system terminates.

$$\frac{C, Z \xrightarrow{c_{pq}!v} C', Z'}{(a, m, C, Z) \xrightarrow{c_{pq}!v} (a, m, C', Z')} \qquad \frac{C, Z \xrightarrow{c_{pq}?v} C', Z'}{(a, m, C, Z) \xrightarrow{c_{pq}?v} (q, m[q \mapsto v], C', Z')}$$

$$\frac{C, Z \xrightarrow{\tau} C', Z'}{(a, m, C, Z) \xrightarrow{\tau} (a, m, C', Z')} \qquad \frac{C, Z \downarrow}{(a, m, C, Z) \downarrow}$$

Figure 7.3: Addition layer of local language operational semantics to integrate coherence model

7.4.3 Endpoint Projection including coherence

The projection of global protocols annotated with coherence information is the earlier projection operator defined in Section 6.3, extended with:

$$(a, m, G) \upharpoonright r \quad = ([a_r \mapsto a], [m_r \mapsto m_{[r]}], G \upharpoonright r)$$

$$(a, m, G) \upharpoonright\upharpoonright R_G = \{r \mapsto (a, m, G) \upharpoonright r \mid r \in R_L\} \qquad \text{if } R_G \subseteq R_L \neq \emptyset$$

Informally, the projection of a global protocol with coherence annotations to a role results in a similar local protocol with a segment of the m annotation holding the attribute managed by the role, and a copy of the a annotation. The projection of an annotated global protocol onto a set of roles is the corresponding group of projections, provided that the set of roles is not empty and there exists a local protocol for at least each role in the set.

Model checking confirms that the above projection is sound and complete. Section 8.2 describes this validation in more detail.

Note that our projection also reveals that not all coherence related information is transferable to local protocols. Individual endpoints are unable to track which attribute has changed most recently, since they lack a system-wide overview of all interactions. As a consequence, endpoints are provided with a local element a that is correct initially, yet will not properly reflect all subsequent state changes.

8

VALIDATION OF THE COHERENCE-EXTENDED CALCULUS

In this chapter we validate our interaction calculus and its ability to support coherence. We use a calculus implementation and model checking. Our final test case is the data grid problem case, formalized in Chapter 5. Verification confirms that the coherence property does not hold for this protocol.

8.1 IMPLEMENTATION OF CALCULUS IN MCRL2

We implement a protocol simulator for our calculus in mCRL2 [5]. See Appendix C for a listing of the source code.

Our protocol simulator, shown in Figure 8.1, is a network of communicating automata. One automaton interpretes the concrete protocol. The remaining automata simulate a system of interacting processes that execute the protocol.

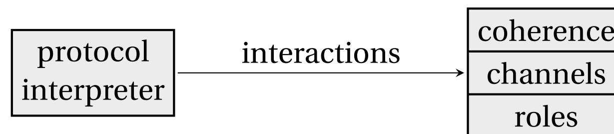


Figure 8.1: Protocol simulator overview

The composition of the simulated system mimics the layered structure of the operational semantics of our calculus. At its foundation are automata that represent an interacting process (role). A next layer consists of automata that model a channel between two participating processes. The top layer is implemented as a single automaton that keeps track of coherence.

Figure 8.2 displays automata that are involved in an example data transfer $c_{ap}.v$. Each row depicts state changes of a single automaton. The automata communicate with each other

via synchronized actions, shown in the same color. We use red to depict synchronized actions related to a `send`, and blue for actions that relate to a `receive`.

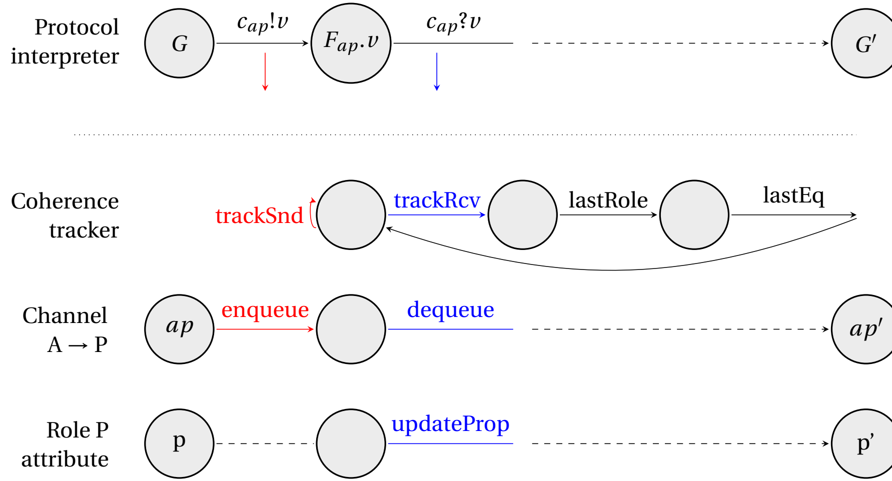


Figure 8.2: Communications between automata, protocol simulator

The *protocol interpreter* automaton interpretes a concrete protocol as a composition of data transfers, in line with the calculus syntax. Each data transfer results in two actions. The automaton first communicates a $c_{ap}!v$ send action and subsequently communicates a $c_{ap}?v$ receive action.

These `send` and `receive` actions are processed by the other automata, the simulated system. Sent data is first captured in a channel queue, and subsequently passed on to a role.

The *channel* automaton ap represents a unidirectional channel from process a to process p. When it synchronizes with the `send` action communicated by the protocol interpreter, it adds the transfered value to its first-in-first-out queue. When it synchronizes with the `receive` action sent by the protocol interpreter, it performs a `dequeue` of the transfered value.

The *recipient role* automaton p is the recipient of the data transfer. It models the participating process p, or more precisely it models the persistent attribute that this process manages. It synchronizes with a `receive` / `dequeue` action, and updates its attribute with the transfered value.

Finally, the *coherence tracker* automaton simulates the coherence annotation layer of our calculus. When it synchronizes with a `receive` action, it subsequently makes two transitions. First, it sends an action `lastRole`, which communicates the name of the attribute that is being changed. Next, it sends an action `lastEq`, which communicates if the updated value of this attribute is equivalent to the other, related attribute. These actions are not synchronized with other actions. They merely report a state. We test for the occurrence of the two actions in our coherence checking formula.

Note that the coherence tracker synchronizes on `send` actions as well. This approach prevents that any new `send` or `receive` can interleave with the processing of a prior `send` or `receive` action by the coherence tracker. As a result, our model checking formula may as-

sume that a `receive` action is always immediately followed by the actions `lastRole` and `lastEq`.

Both global protocols and local protocols are supported by the simulator. Our program includes an explicit implementation for the term `transfer` used in the global calculus and the actions `send` and `receive` that feature in both the global and the local calculus. All other constructs are implemented using native mCRL2 functions. This includes the action `tau`, as well as the compositions sequential, choice, recursion, and parallel.

8.2 Validation of projection from global to local calculus

We use model checking to validate the projection of our interaction calculus to the local endpoint calculus. The designed projection is sound and complete if a labeled transition system specified using a global interaction protocol, is weakly bisimilar to a system composed of projected endpoints and channels [16]. This requires that the two systems will abide by the same rules and transition in sync. The adjective *weak* expresses that the requirement does not apply to τ transitions which are entirely internal to a process.

We model check bisimularity using two protocol samples. Apart from the concrete protocol specification, all our simulations are based on the same simulation program code. All simulations will include relevant channels and the coherence tracker.

First, we sample a 'good' protocol G_g. The syntax $C(a, p, one)$ is used by the protocol simulator program to implement a transfer $c_{ap}.v$ where $[v \mapsto one]$. In global protocol G_g, role a transfers either the value one or the value two to role p. Once this has been done, and regardless of the value received, role p communicates either the value one or two to role q.

$$G_g = (C(a, p, one) + C(a, p, two)) \cdot (C(p, q, one) + C(p, q, two));$$

We project[11] this global interaction to a parallel composition L_g of three endpoint protocols. Upon a data transfer, roles will execute either a `send` action, or a `receive` action, or not participate in the interaction by performing a `tau` action.

$$L_{ga} = (\text{send}(a, p, one) + \text{send}(a, p, two));$$
$$L_{gp} = (\text{receive}(a, p, one) + \text{receive}(a, p, two)).$$
$$(\text{send}(p, q, one) + \text{send}(p, q, two)) \cdot \text{tau};$$
$$L_{gq} = \text{tau} \cdot (\text{receive}(p, q, one) + \text{receive}(p, q, two));$$
$$L_g = L_{ga} \mid\mid L_{gp} \mid\mid L_{gq};$$

The output of the mCRL2 bisimulation checking tool `ltscompare` confirms that the corresponding labeled transition systems are weakly bisimilar.

[11] We use the projection operator documented in Sections 6.3 and 7.4.3.

Our second sample involves a counter-example G_b and its projection L_b:

$G_b = C(a, p, one) \cdot C(b, q, one);$

$L_{ba} = \text{send}(a, p, one) \cdot \text{tau};$

$L_{bb} = \text{tau} \cdot \text{send}(b, q, one);$

$L_{bp} = \text{receive}(a, p, one) \cdot \text{tau};$

$L_{bq} = \text{tau} \cdot \text{receive}(b, q, one);$

$L_b = L_{ba} || L_{bb} || L_{bp} || L_{bq};$

G_b is an example of a bad protocol, as its specification cannot be followed through correctly by endpoints. The protocol specifies that role b may start a communication only after the communication between roles a and p has completed. However, role b is not involved in the prior communication and therefore it lacks the relevant knowledge to time its own communication.

Model checking confirms that the labeled transition systems for G_b and L_b are not (weakly) bisimilar.

8.3 Validation of coherence expressiveness

We use the protocol simulator program to validate that we can reason about coherence using the combination of 1) our calculus and 2) our coherence definition.

We note that the mCRL2 model checker offers limited support for checking state variables and no support at all for formulas stated in CTL. The coherence tracker function in our simulation program resolves the first limitation by reporting coherence state information as actions (`lastRole` and `lastEq`). We transpose the coherence property CTL formula to a μ-calculus based version, to resolve the second limitation. The resulting formula is listed in Appendix D.

Using the mCRL2 tools `lts2pbes` and `pbes2bool`, we verify that the property complies with the expected result from transitions listed in the Kripke structure in Figure 7.1. In addition, we verify test cases that involve sequential, choice, recursion and parallel compositions. These test cases are listed in Appendix E.

Finally, we test our data grid problem use case:

$G = C(a, p, one) \cdot C(a, q, one) \; || \; C(b, p, two) \cdot C(b, q, two);$

In our test protocol, a and b are agents, p manages a data object, and q manages a replica. Agent a sends the value one whereas agent b sends value two. Verification confirms that this concrete protocol does not protect coherence.

9

PROTOTYPE LANGUAGE EXTENSIONS THAT PROTECT COHERENCE

In the previous chapters, we designed and implemented a calculus. Using this calculus, we are able to detect if a protocol meets coherence requirements. We will now discuss how this calculus can be used to *protect* coherence. Our prototype implementation adds (un)lock operations to the calculus, and deploys a strategy named *chained-coherence*.

9.1 TOWARDS A STRATEGY FOR COHERENCE PROTECTION

Ordering data operations is key to data coherence protection, similar to data consistency protection. In both cases, the desired property is potentially endangered by interleaved processing of data. This threat is mitigated by requiring that the execution of data operations meets an agreed upon order.

The inevitable balance between consistency and other desired properties of distributed systems has lead to multiple consistency models with different ordering requirements [30]. With regard to coherence, we propose to use the ordering requirements as specified in the coherence property Definition 7.3.3. These requirements implement a sequential consistency model for coherence.

While a protocol specification may enforce some ordering in interactions, this will in general not be sufficient to protect coherence. Our data grid problem case is an example of a protocol that does not protect coherence as a result of interleaved interactions. Note that even if a protocol itself does not include interleaved operations, the interleaved execution of multiple instances of such a protocol might introduce risks to data coherence.

A contract between processes can be used to protect coherence. For instance, we may consider to add a mutual exclusive access mechanism to our calculus so that processes can synchronize their operations. Alternatively, the mechanism could be implemented on a service level. For example, a coordinating server agent might choose to only accept one

connection at a time, effectively disallowing interleaved execution of the protocol. Clearly, this alternative approach may introduce significant throughput limitations.

We will prototype a solution to our coherence problem that builds on our calculus extended with `lock` and `unlock` operations. Locking all attributes could be considered to enforce exclusive access in a scenario where we protect coherence between two attributes. Scenarios that involve more related attributes may require a more sophisticated strategy, which we will discuss next.

9.2 Throughput considerations for coherence

Whenever attributes need to remain coherent, this limits the throughput of update operations on those attributes. The throughput limitation stems from the requirement that attributes may not be updated in isolation. All attributes need to be updated to the same value, before a subsequent request can be processed. This can be a time-consuming operation when the attributes are managed by geographically distributed processes that interact via a high-latency network. Also note that the size of an attribute set will have a negative impact on throughput.

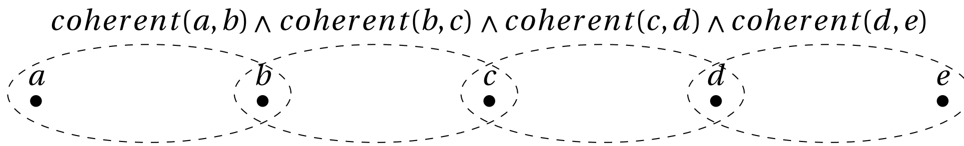


Figure 9.1: Chained-coherence with five attributes

We propose a *chained-coherence* strategy to increase the throughput for large coherent sets. This implementation strategy involves the deployment of linked subsets of coherent attributes instead of processing a single large set. For instance, in order to meet the requirement $coherent(a,b,c)$ one might consider to only implement $coherent(a,b)$ and $coherent(b,c)$. Since attribute b is part of both subsets, any update to b must also be applied to both a and c and vice versa. The chained-coherence strategy guarantees that update requests will be applied to all attributes, in the same order. In contrast to using a single large set, the approach using linked subsets does not prohibit any processing of a subsequent request until an update has been applied to all attributes.

a	b	c	d	e
0	0	0	0	0
1	0	0	0	0
1	1	0	0	0
2	1	1	0	0
2	2	1	1	0
3	2	2	1	1

Figure 9.2: Single-chain approach

a	b	c	d	e
0	0	0	0	0
0	0	1	0	0
0	1	1	1	0
1	1	2	1	1
1	2	2	2	1
2	2	3	2	2

Figure 9.3: Two-chain approach

There are many options for the implementation of a coherence-chain strategy. We will use an example with five attributes as outlined in Figure 9.1 to discuss a single-chain and a

two-chain implementation.

Figure 9.2 shows state changes resulting from a single-chain (a-b-c-d-e) approach. Each row represents a state with the current value per attribute. First, attribute a is updated with the new value 1, then this change is applied to b, followed by c then d and finally e. The fourth row demonstrates how attribute a is free to apply a next update request (value 2) as soon as b is on par with the previous request, resulting in a processing pipeline.

Figure 9.3 showcases an approach with two chains: c-b-a and c-d-e. First, attribute c is updated. Subsequently, the chains can be updated in parallel without the risk of endangering coherence. As soon as b and d both have applied the update, then attribute c is free to process a next update request. This approach is slightly more involved than the single-chain, yet results in a higher throughput. The last row shows how a second update request has been applied to four attributes, compared to two attributes in the single-chain approach.

9.3 Prototype with coherent update

Can we extend our calculus syntax with an operation that protects coherence? We add a new operation CohUpd that implements chained-coherence with a single-chain. We change our data grid problem test case to use this operation:

$G = \text{CohUpd}(a, [p, q], one) \mid\mid \text{CohUpd}(b, [p, q], two);$

Our solution is based on the calculus designed in Chapter 7, extended with `CohUpd`, `lock`, and `unlock` operations. A role that executes a `lock` operation will block until the lock is acquired. The actual protocol simulator source code changes and additions are listed in Appendix F. Conceptually, the pseudo code listed in Figure 9.4 summarizes our protocol

```
CohUpd(fromRole, toList, value) {
  n = length(toList);
  for (i = 0; i < n; i++) {
    lock(i);
    if (i > 1) then
      unlock(i-2);
    transfer(fromRole, i, value);
    ack(i, fromRole);
  }
  unlock(n-2);
  unlock(n-1);
}
```

Figure 9.4: Pseudo code for coherent update operation

simulator implementation of the coherent update operation. The `CohUpd` operation iterates through the targeted destinations in a fixed order. It acquires a lock on the attribute managed by the target role, then transfers the new value to the target role, and waits for acknowledgement that the role has completed an update of its attribute. The oldest lock is released, but only if at least two attributes remain locked. After all target roles have been

visited, the remaining locks are released.

Model checking confirms that the coherence property indeed holds for this protocol. Note that our prototype implements the locking mechanism provisionally as a responsibility of the process managing the attribute. While this seems a promising approach, further research will be needed to investigate how our calculus should incorporate mutual exclusive access.

10

Discussion and conclusions

10.1 Discussion

The multiparty asynchronous session types theory allows us to reason over interactions between processes. We build on this theory to establish how data coherence is affected by interactions.

The original MPST calculus, designed by Honda et al., introduces constructs and a type discipline to structure interactions between processes [14]. We provide similar constructs to structure interactions, yet we focus on concrete values that are being exchanged rather than typed messages. Montesi note that the general idea to specify process interactions in a global program and subsequently project this program to endpoint nodes constitutes a new paradigm, which they name choreographic programming [23].

Jongmans and Yoshida have enhanced the MPST calculus by Honda et al. to make it more expressive [16]. Their calculus supports both synchronous and asynchronous communications. Asynchronous communications are supported implicitly by modeling channels as additional roles. Cruz-Filipe and Montesi argue that encoding asynchronous communications using roles simplifies the grammar [9]. In contrast, we provide explicit support for (only) asynchronous communications. Our approach bears the advantage that communication actions can be treated uniformly with regard to coherence annotations. For instance, in our calculus a `receive` action will always pertain to a process receiving data, it will never relate to a channel that receives a value to be queued.

Based on the CAP theorem, we work from the assumption that coherence in a non-transactional distributed system will require a compromise between consistency and availability [11]. As a consequence, our definition of coherence uses an equivalence relationship that is *almost* invariant. When an attribute's value changes, related attributes are allowed to temporarily carry the original value, until they are updated as well. Clearly, a more strict definition would be feasible, if we can restrict our context to transactional distributed systems. For instance, Spillane et al. have researched how transactional access can be en-

abled for subsystems that do not natively support transactions [25]. While it was possible to add transaction support to a POSIX based file system, their conclusion is that adding transaction support mechanisms is not easy to accomplish efficiently in existing operating systems.

We have validated our research using a limited set of samples that cover most of the syntactical compositions offered by our calculus. Some compositions are difficult to validate though. In particular, mCRL2 does not support scenarios that involve a combination of interleaving and recursion. Instead, we have validated these constructs in isolation.

Our research covers an attribute update operation only. While this is a common and fundamental operation, distributed systems such as a data grid provide other functions that have remained out of scope. For instance, moving a collection tree with multiple data objects to a new location as a compound operation is an example of a scenario that may involve a nested form of coherence.

The chained-coherence strategy has been tested under simulator conditions. Further benchmarking experiments will be needed to assess the actual impact on throughput performance and to benchmark these results against alternative approaches.

10.2 Conclusions

Our research provides a method that can be used to prove that data remains coherent in the context of a non-transactional distributed system. For instance, this method could be applied to interactions of a data grid. Concrete interactions specified in a domain specific language based on our calculus can be analyzed statically to verify that they meet coherence requirements. As suggested by our prototype, our calculus can be used as a foundation to develop interactions that protect coherence.

Our method can be applied to analyze interactions in existing systems using protocol simulation. We have demonstrated this capability by successfully applying the method to an existing data grid issue.

We recommend to use *linked coherent subsets* in scenarios where coherence is required for a large set of attributes. Our research suggests that throughput performance may benefit from using linked subsets over a single large set. The related chained-coherence strategy requires that all processes agree on processing the data in the same order.

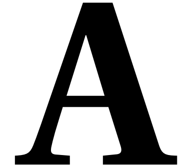
10.3 Future work

How well can we specify all data grid process interactions using a choreographic programming paradigm? We have assumed an ideal interaction context where messages are communicated without exception and processes are able to update their attributes in a proper and timely fashion. In our prototype, we have combined choreographic programming with mutual exclusive access mechanisms. More research will be needed to establish how our

calculus can facilitate exception handling and negotiation between processes.

Next to protection of coherence, research is needed to investigate how choreographic programming methods could be used to specify how to *reintroduce* coherence after it is lost. For instance, Cherrier and Ghamri-Doudane provide a method for coherence fault-recovery in the context of an unreliable Internet Of Things network infrastructure [6]. How can we cater for fault recovery in our calculus?

We have concluded that coherence is a system-level property. Our current method does not provide endpoint nodes with sufficient state information needed to make informed decisions related to coherence. Prior research, for instance Lloyd et al., shows that endpoint nodes can be provisioned with state information by using data versioning annotations [20]. Can we combine these methods with choreographic programming to efficiently manage coherence at the endpoint node level?



SOFTWARE COMPONENTS USED IN RESEARCH

The following software components have been used in the context of this research:

- Antlr version 3.5.2
- Eclipse Photon (4.8.0)
- iRODS 4.2.6
- Java-8-openjdk (JavaSE-1.8)
- Linux Debian Buster 4.19
- Maven 3.6.0
- mCRL2 toolset 202106.0
- OpenJDK Runtime Environment (build 11.0.7+10-post-Debian-3deb10u1)
- Scribble release-0.4.3 (supports par syntax)
- Scribble[12] at release-0.4.3 with additional changes (last git commit 13-May-2019, 53e520a8cd916bfadd20db947568f676210d350c) (much improved, yet does not support par syntax)
- UPPAAL 4.1.22
- UPPAAL Tiga 4.1.4

The software created during this research, (UPPAAL model and mCRL2 protocol simulator) is open access available via DOI 10.5281/zenodo.6366593

[12] See https://github.com/scribble/scribble-java/

B

UPPAAL MODEL OF PROBLEM USE CASE

Below is a partial listing of the UPPAAL source code used to model the problem use case and prototypes.

The global declaration section shown in Figure B.1 provides details on identifiers used in diagrams discussed in Section 5.2. Local declarations for templates, not shown here, are limited to the declaration of a variable used for iteration, if any.

```
// A is number of agents that concurrently modify a data object property
const int A = 3;
typedef int[0,A-1] id_a; // agent id's

// R is number of replicas that a data object has
const int R = 3;
typedef int[0,R] id_obj; // objects (both data object and replicas)
typedef int[1,R] id_r; // replicas only

// global state, property of the dataobject and its replicas
// agents update the data/replica object with the value representing their id
id_a objValue[id_obj]; // current persistent value held at Object (update is applied)
id_a objRead[id_obj]; // last value received via Read by Object (update is read)

// communication channels between agents and objects
chan upd[id_obj], ack[id_obj];
meta id_a updValue;     // the value that the object's property will be set to

// communication channels between agents and locks
const int MUTEX = 0;  // first object's mutex also used for generic mutex lock
chan lock[id_a][id_obj], unlock[id_a][id_obj];  // one mutex per object
id_a currentLock[id_obj];
```

Figure B.1: Global declarations for problem use case model in UPPAAL

C

PROTOCOL SIMULATOR IN MCRL2

```
sort RoleName = struct a|b|p|q|r;
sort Value = struct zero|one|two|three|four|ack;
sort SyncStates = struct first|second|more;

act  send,enqueue,trackSend,send',
     receive,dequeue,updateProp,trackReceive,
        receive' : RoleName # RoleName # Value;
     lastRole : RoleName;
     lastEq : Bool;

proc Role'(N:RoleName,prop:Value) =
     sum from:RoleName, v:Value . ((N != from) -> updateProp(from,N,v).Role'(N,v));

     Channel'(from,to:RoleName,data:List(Value),size:Int) =
     sum v:Value . ((size < 5) -> enqueue(from,to,v).
         Channel'(from,to,data <| v,succ(size)) ) +
     (size > 0)               -> dequeue(from,to,head(data)).
         Channel'(from,to,tail(data), pred(size) );
```

Figure C.1: Protocol simulator model, part 1

```
   Coherence'(coh1,coh2:RoleName,coh1val,coh2val:Value,last:RoleName) =
   sum from,to:RoleName,v:Value .
   (to == coh1) -> (trackReceive(from,to,v).
           lastRole(to).lastEq(v==coh2val).
           Coherence'(coh1,coh2,v,coh2val,to) ) +
   sum from,to:RoleName,v:Value .
   (to == coh2) -> (trackReceive(from,to,v).
           lastRole(to).lastEq(v==coh1val).
           Coherence'(coh1,coh2,coh1val,v,to) ) +
   sum from,to:RoleName,v:Value .
   ((to != coh1) && (to != coh2)) -> (trackReceive(from,to,v).
           lastRole(to).lastEq(coh1val==coh2val).
           Coherence'(coh1,coh2,coh1val,coh2val,to) ) +
   sum from,to:RoleName,v:Value . (trackSend(from,to,v).
           Coherence'(coh1,coh2,coh1val,coh2val,last) ) ;

% Data transfer 'Cpq' is a basic construct of our global language:
 C(from:RoleName,to:RoleName,v:Value) = send(from,to,v).receive(from,to,v);


% ---------- a testset of concrete instances of coherence model protocol
 G0 = C(a,p,one).C(a,q,one);   % coh(p,q) = true   (s0,s1,s3)

% initialization
 Role(N:RoleName) = Role'(N,zero);
 Chan(from,to:RoleName) = Channel'(from,to,[],0);
 Coherence(r1,r2:RoleName) = Coherence'(r1,r2,zero,zero,a);

init allow( { send', receive', lastRole,lastEq},
   comm( {send|enqueue|trackSend -> send',
      receive|dequeue|updateProp|trackReceive -> receive'},
     Role(a) || Role(b) || Role(p) || Role(q) || Role(r) ||
     Chan(a,p) || Chan(a,q) || Chan(a,r) || Chan(b,p) || Chan(b,q) ||
     Chan(b,r) || Chan(p,q) || Chan(p,r) || Chan(q,p) || Chan(q,r) ||
     Chan(r,p) || Chan(r,q) || Coherence(p,q) ||
     G0   ) );
```

Figure C.2: Protocol simulator model, part 2

D

COHERENCE PROPERTY TRANSPOSED TO μ-CALCULUS

```
% Our definition of coherence in CTL:
%     coherence(p,q)    = follows(p,q) AND follows(q,p)
%         follows(p,q) = last(p) AND (p!=q) => AX(
%                              A(not last(p) AND not last(q) U last(q) AND (p==q))   )

% Implementation of the above CTL formula in mcrl2 muCalculus syntax:
% part 1: follows(p,q)
% part 2: follows(q,p)

(
   % part 1:
   [true*.lastRole(p).lastEq(false)]     (  % 1a: AG(last(p) AND p != q  => ...... )
      [!lastRole(q)*.lastRole(p)] false     % 1b: A(not last(p) U last(q)
       &&                                    %     NB: remainder of Until clause
                                             %         is checked as part of 1c
                                             % 1c: A(not last(q) U last(q) AND (p==q))
      [!lastRole(q)*.lastRole(q).lastEq(false)] false
   )
&&
   % part 2:
   [true*.lastRole(q).lastEq(false)]     (
          [!lastRole(p)*.lastRole(q)] false   &&
          [!lastRole(p)*.lastRole(p).lastEq(false)] false
   )
)
```

Figure D.1: Coherence property implementation in μ-calculus

E

TEST CASES PROTOCOL SIMULATOR

```
% test of basic transitions as shown in figure Kripke structure
% (excluding mirrored cases)
% s0..s6 refer to states in figure Thesis report
  G0 = C(a,p,one).C(a,q,one);                        % coh(p,q) = true   (s0,s1,s3)
  G1 = C(a,p,one).C(a,q,two);                        % coh(p,q) = false  (s0,s1,s4)
  G2 = C(a,p,one).C(a,p,two);                        % coh(p,q) = false  (s0,s1,s6)

% tests of data grid use case p = data object, q = replica
  G3 = C(a,p,one).C(a,q,one)||C(b,p,two).C(b,q,two); % coh(p,q) = false  (race)
  G4 = C(a,p,one).C(a,q,one).C(b,p,two).C(b,q,two)+
       C(b,p,two).C(b,q,two).C(a,p,one).C(a,q,one);  % coh(p,q) = true
%                                                  (different protocol solves race)

% other testcases that exercise language constructs and model checking formula

  % interleaved update of non-related attribute
  G10= C(a,p,one).C(a,r,one).C(b,q,one);             % coh(p,q) = true
  G11= C(a,p,one).C(a,r,one).C(a,q,two);             % coh(p,q) = false

  % choice constructs
  G12= C(a,p,one).C(a,r,one).
       (C(a,q,one) + C(a,p,two));                    % coh(p,q) = false
  G13= C(a,p,one).C(a,r,one).
       (C(a,q,one) + C(a,r,two));                    % coh(p,q) = true

  % parallel construct
  G14= C(a,p,one).C(a,q,one) || C(b,r,two);          % coh(p,q) = true

  % recursive construct
  G15= C(a,p,one).C(a,q,one).
       C(a,q,two).C(a,p,two).G15;                    % coh(p,q) = true
```

Figure E.1: Test cases for protocol simulator

F

PROTOCOL SIMULATOR EXTENSIONS TO SUPPORT COHERENT UPDATE

```
% our CohUpd keeps state via SyncStates
sort SyncStates = struct first|second|more;

% locking actions added
act  lock,lock',unlock,unlock' : RoleName # RoleName;

% Role' updated to also respond to (un)lock requests
proc Role'(N:RoleName,prop:Value,locked:Bool,holder:RoleName) =
     sum from:RoleName, v:Value . ((N != from) ->
               updateProp(from,N,v).Role'(N,v,locked,holder)) +
     % unlock
     sum requester:RoleName . ( (locked && (requester==holder)) ->
               unlock(requester,N).Role'(N,prop,false,requester)) +
     % lock
     sum requester:RoleName . ( (!locked) ->
               lock(requester,N).Role'(N,prop,true,requester)) ;

% Role needs to also maintain lock state
     Role(N:RoleName) = Role'(N,zero,false,a);
```

Figure F.1: Protocol simulator cohUpd extensions, part 1

```
% CohUpd and CohUpd' added

  CohUpd(from:RoleName, toList: List(RoleName), v: Value) =
     ((#toList > 1) && !(from in toList)) ->
        % at least 2 roles coherent, 'from' must not be one of them
        CohUpd'(first, from, toList, v, []);

  CohUpd'(syncState: SyncStates, from:RoleName, todoList: List(RoleName),
          v: Value, doneList: List(RoleName)) =
     (syncState == first)  -> (lock(from, head(todoList)).
           C(from, head(todoList), v).C(head(todoList), from, ack).
           CohUpd'(second, from, tail(todoList), v, doneList<|head(todoList) )) +
     (syncState == second) -> (lock(from, head(todoList)).
           C(from, head(todoList), v).C(head(todoList), from, ack).
           CohUpd'(more, from, tail(todoList), v, doneList<|head(todoList)) ) +
     ((syncState == more) && (todoList != [])) -> (lock(from, head(todoList)).
           C(from, head(todoList), v).C(head(todoList), from, ack).
           unlock(from, head(doneList)).
           CohUpd'(more, from, tail(todoList), v, tail(doneList)<|head(todoList) )) +
     ((syncState == more) && (todoList == []) ) ->
          ((doneList != []) ->
                ( unlock(from, head(doneList)).
                CohUpd'(more, from, [], v, tail(doneList)) ))   ;

% locking actions added
init
  allow(
     { send', receive', lastRole, lastEq, lock', unlock'},
     comm(
        {send|enqueue|trackSend -> send',
         receive|dequeue|updateProp|trackReceive -> receive',
         lock|lock -> lock', unlock|unlock -> unlock'
        },
```

Figure F.2: Protocol simulator cohUpd extensions, part 2

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Sung-Shik Jongmans, who inspired and supported me during this research, and who provided me with many valuable insights and ideas. Sung showed me how research of formal languages can be an exciting, enjoyable and worthwhile journey and I feel blessed that he was my supervisor.

I appreciate the friendship and support of my fellow student Roel Erps. We have been buddies, sharing our highs and lows, during the entire (pre)master, working together on many course assignments.
Further, I like to thank Sietse Snel, for taking the time to review and comment on an early draft of my thesis. I hope to be able to return the favor.

Last but not least, I wish to dearly thank Tineke, my loving partner forever, whose love and support fuels all my endeavors.

BIBLIOGRAPHY

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995. ISSN 1432-0452. doi: 10.1007/BF01784241. URL https://doi.org/10.1007/BF01784241. 8

[2] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL—a tool suite for automatic verification of real-time systems. In *International hybrid systems workshop*, pages 232–243. Springer, 1995. 15, 17, 21

[3] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77 – 121, 1985. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(85)90088-X. URL http://www.sciencedirect.com/science/article/pii/030439758590088X. 26

[4] Stefan Blom, Jaco van de Pol, and Michael Weber. LTSmin: Distributed and symbolic reachability. In *International Conference on Computer Aided Verification*, pages 354–359. Springer, 2010. 17

[5] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 Toolset for Analysing Concurrent Systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17465-1. 17, 36

[6] Sylvain Cherrier, Yacine M Ghamri-Doudane, Stéphane Lohier, and Gilles Roussel. Fault-recovery and coherence in internet of things choreographies. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 532–537. IEEE, 2014. 46

[7] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187 – 200, 2000. ISSN 1084-8045. doi: https://doi.org/10.1006/jnca.2000.0110. URL http://www.sciencedirect.com/science/article/pii/S1084804500901103. 4, 5, 6

[8] Edmund M Clarke Jr, Orna Grumberg, and Doran A Peled. *Model checking*. MIT Press, Cambridge, Massachusetts, 1999. 32

[9] Luís Cruz-Filipe and Fabrizio Montesi. Encoding asynchrony in choreographies. In *Proceedings of the Symposium on Applied Computing*, pages 1175–1177, 2017. 44

[10] Wan Fokkink. *Introduction to Process Algebra*. Springer Berlin Heidelberg, 2000. doi: 10.1007/978-3-662-04293-9. URL https://doi.org/10.1007/978-3-662-04293-9. 26

[11] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL https://doi.org/10.1145/564585.564601. Place: New York, NY, USA Publisher: Association for Computing Machinery. 2, 44

[12] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. Publisher: ACM New York, NY, USA. 7

[13] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-69722-0. 9, 10

[14] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, 2008. 3, 10, 15, 44

[15] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling Interactions with a Formal Foundation. In Raja Natarajan and Adegboyega Ojo, editors, *7th International conference on Distributed Computing and Internet Technology*, pages 55–75, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19056-8. 10, 15, 19, 20

[16] Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In *European Symposium on Programming*, pages 251–279. Springer, Cham, 2020. 16, 17, 25, 27, 29, 38, 44

[17] Saul A Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963. 16, 32

[18] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. 7, 8

[19] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986. Publisher: Springer. 2

[20] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011. 8, 46

[21] David Mazieres and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117, 2002. 8

[22] Robin Milner. The polyadic pi-calculus: a tutorial. In *Logic and algebra of specification*, pages 203–246. Springer, 1993. 10

[23] Fabrizio Montesi. *Choreographic programming.* PhD thesis, University of Copenhagen, Denmark, 2013. 44

[24] B.C. Pierce, B. C, Coutts Information Services, and M. I. T. Press. *Types and Programming Languages.* The MIT Press. MIT Press, 2002. ISBN 978-0-262-16209-8. 10

[25] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 29–42, USA, 2009. USENIX Association. event-place: San Francisco, California. 45

[26] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms.* Prentice-Hall, 2nd edition edition, 2007. ISBN 0-13-239227-5. 1, 2, 3

[27] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149. IEEE, 1994. 7, 8

[28] Francisco J Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed consistency for shared distributed objects. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 163–172, 1999. 7

[29] Frits Vaandrager. A First Introduction to Uppaal. *Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook*, pages 18–48, 2011. 18, 23

[30] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49(1):1–34, 2016. Publisher: ACM New York, NY, USA. 7, 40

[31] M. Wan, W. Schroeder, A. Rajasekar, and R. Moore. Universal view and open policy: Paradigms for collaboration in data grids. In *Collaborative Technologies and Systems, International Symposium on*, pages 322–329, Los Alamitos, CA, USA, May 2009. IEEE Computer Society. doi: 10.1109/CTS.2009.5067497. URL https://doi.ieeecomputersociety.org/10.1109/CTS.2009.5067497. 5, 6

GLOSSARY OF TERMS AND ABBREVIATIONS

algebra
A generalization of arithmetic operations. It consists of mathematical symbols and rules for manipulating them. For instance $c = a + b$ involves the symbols $\{a, b, c, =, +\}$.

bisimulation
A binary relation between two state transition systems where the systems behave in an equivalent way so that one system effectually "simulates" the behavior of the other system and vice versa. In concurrent systems modeling, bisimulation is used to express the assumption that all processes will abide by the same rules and transition in sync.

calculus
A method of computation by reasoning over symbols. It specifies how one symbol can be derived from other symbols in a transition step. For instance $f(x) = x^2$ shows how x can transition to its square. A calculus may consist of a grammar to specify the symbols and operational semantics to specify rules for valid transitions between compositions of symbols.

coherence
Invariant content relationship between two or more data items. See Section 7.3 for a formal definition.

data grid
Integrated architecture that supports access to and management of distributed data.

data object
Name to reference the set of instances of a data file in a data grid.

grammar
A grammar defines the compositional structure (syntax) of a language. It does so by specifying valid compositions (strings) using an alphabet.

LTS Abbreviation for Labeled Transition System.

MPST
Abbreviation for Multi-Party asynchronous Session Types.

structural operational semantics
A set of rules that specify valid sequential steps for processing instances of a language. For example a rule could specify that the expression $A + B$ must be evaluated before the asignment to C in the event of a statement $C = A + B$. Commonly the rules ensure a deterministic form of processing. Hence operational semantics define the (sequence of) interpretation of language compositions.

process calculus
A calculus to model concurrent systems by specifying how processes can interact.

protocol
Formal specification of a session.

replica
Name to reference a single instance of a file in a data grid.

resource
Name to reference a particular storage medium in a data grid.

session
Structured interaction between two or more processes.

SOS Abbrevation of Structural Operational Semantics.

weakly bisimilar
A bisimulation that is insensitive to and allows for *internal* actions, actions that do not involve an interaction.