Tool Supported Specification and Verification of Highly Available Applications

Dissertation

Eingereicht am Fachbereich Informatik
der Technischen Universität Kaiserslautern

von

Peter Zeller

September 18, 2020

Abstract

Today, information systems are often distributed to achieve high availability and low latency. These systems can be realized by building on a highly available database to manage the distribution of data. However, it is well known that high availability and low latency are not compatible with strong consistency guarantees. For application developers, the lack of strong consistency on the database layer can make it difficult to reason about their programs and ensure that applications work as intended.

We address this problem from the perspective of formal verification. We present a specification technique, which allows specifying functional properties of the application. In addition to data invariants, we support history properties. These let let us express relations between events, including invocations of the application API and operations on the database.

To address the verification problem, we have developed a proof technique that handles concurrency using invariants and thereby reduces the problem to sequential verification. The underlying system semantics, technique and its soundness proof are all formalized in the interactive theorem prover Isabelle/HOL. Additionally, we have developed a tool named Repliss which uses the proof technique to enable partially automated verification and testing of applications. For verification, Repliss generates verification conditions via symbolic execution and then uses an SMT solver to discharge them.

Zusammenfassung

Informationssysteme werden mittlerweile oft als verteilte Systeme gebaut, um hohe Verfügbarkeit und geringe Latenz zu erreichen. Zur Realisierung solcher Systeme kann auf hochverfügbare Datenbanken zurückgegriffen werden, welche die Verteilung der Daten übernehmen. Allerdings ist auch bekannt, dass hohe Verfügbarkeit und geringe Latenz nicht kompatibel mit starken Garantien bezüglich Datenkonsistenz sind. Für Anwendungsentwickler kann dieser Verzicht auf starke Konsistenz es erschweren, alle Vorgänge im System präzise zu verstehen und somit sicherzustellen, dass Anwendungen wie gewünscht arbeiten.

Wir gehen dieses Problem von der Seite der formalen Verifikation von Software an. Dazu stellen wir eine Spezifikationstechnik vor, die es erlaubt, funktionale Eigenschaften einer Anwendung zu spezifizieren. Neben Invarianten auf den Daten, erlaubt diese Technik auch die Formulierung von History-Eigenschaften. Diese erlauben es, bestimmte Ereignisse aus dem Ausführungsverlauf einer Anwendung miteinander in Bezug zu bringen. Zu den Ereignissen gehören Aufrufe der Anwendungsschnittstelle und Operationen, die auf der Datenbank ausgeführt wurden.

Um das Verifikationsproblem zu lösen, haben wir eine spezielle Beweistechnik entwickelt, welche den Aspekt der Nebenläufigkeit mithilfe von Invarianten behandelt und damit das Problem auf den sequentiellen Fall reduziert. Die zugrundelegende Semantik des Systems, die Beweistechnik und der Beweis dessen Korrektheit sind im interaktivem Theorembeweiser Isabelle/HOL formalisiert. Desweiteren haben wir ein Programmierwerkzeug namens Repliss entwickelt, welches die Beweistechnik verwendet, um die Verifikation und das Testen von Anwendungen teilweise zu automatisieren. Zur Verifikation verwenden wir eine Form der symbolischen Auswertung, welche mathematische Beweisverpflichtungen erzeugt, die dann durch einen SMT-Solver bewiesen werden können.

Contents

List of Figures

Chapter 1

Introduction

It is challenging to build information systems that are highly available, fast, and correct. The design space obviously is huge, but some constraints do apply: To achieve high availability, one cannot rely on a single physical machine. Even relying on several machines in the same location means that a single regional event can render the system unavailable. Therefore, high availability requires to replicate the system at several locations.

When we look at the aspect of performance, we also derive at the same conclusion, albeit for different reasons. If we want low latency in an interactive application, the processing must take place close to the client so that the effect of network latencies is minimized. For an application with clients around the globe this requires geo-distribution, i.e., the deployment of different machines close to clients around the globe. For applications with unreliable connectivity, for example in mobile applications, it might even be necessary to have a working copy of the data on the local device.

When it comes to correctness, developers must decide which correctness guarantees they want to provide to clients and which consistency level they in turn require from the infrastructure they use. Both can be split into two classes: properties that require synchronous communication with a quorum of machines and properties that can be fulfilled without synchronous communication. For example linearizable registers cannot be implemented without synchronization, as stated by the CAP theorem [GL02]. Weaker consistency models like causal consistency [Llo+11] can be achieved without synchronization as it poses no restriction on the staleness of data. If we want high availability in the sense that every node should be able to respond to queries independently of the state of other nodes, and low latency in the sense that we only want to pay the cost of the delay to the closest available node, then we are required to choose a model without synchronized operations and therefore with weaker consistency.

Obviously, it becomes harder to build a correct application when the underlying infrastructure provides fewer guarantees. Therefore, many application developers choose a strongly consistent backend for their application and thus loose the benefits of high availability and low latency. Stemming from this

situation, we have to ask: Is there a set of rules or guidelines that would allow developers to ensure the correctness of their applications even under weak consistency?

In this thesis we address this question from a formal perspective, asking how we can formally verify the correctness of an application that runs on infrastructure with weak consistency guarantees. More precisely, we consider deductive software verification [Fil11], where the correctness of a program is specified in a formal language. A set of logical rules then allows to combine the specification and the program and derive mathematical statements, such that the program is correct if the mathematical statements are correct. These statements are called verification conditions and are automatically generated from the specification and the program text through a process called *verification condition generation*.

Existing techniques and tools for deductive verification are not directly applicable to applications with concurrency and weak consistency (see Section 3.3 for related work). If they support concurrency, they usually handle access to shared memory via mutual exclusion, which requires synchronization and therefore is not applicable in our scenario.

We therefore have developed our own proof technique, which reduces the problem of verifying a concurrent application with weak consistency down to the verification problem of a sequential program. This problem can then be tackled using existing techniques and tools. The reduction is possible, because we restrict the architecture and expressiveness of our invariants such that concurrently running processes are mostly independent. The remaining points where processes can interfere with each other are handled by user defined invariants and generic history invariants that are true for all programs or can be automatically derived from the program.

An aspect that distinguishes our approach from many other verification techniques lies in the specification language. In a sequential setting, effects of program parts like procedures can be specified by relating the pre- and the post-state using pre- and post-conditions. However, this is difficult when the state can also be affected by other, concurrently running, processes. We address this challenge by integrating events into our specification language. These events include low level operations on the underlying infrastructure, as well as higher level events of clients interacting with the application. Thus, we can describe the effect of procedures based on events instead of the shared database state, which avoids the problem of concurrent changes. After all, the structure of events is mostly immutable (the past cannot change). The event-oriented approach also makes specifications more expressive than purely state based ones.

1.1. Contributions

This thesis develops a tool supported, formalized proof technique for highly available applications. The thesis provides the following contributions:

1. A specification technique for highly available applications based on history invariants.
2. A proof technique for highly available applications, which has been fully formalized in Isabelle/HOL.
   a) We use a small-step operational semantics to describe the system. The semantics is parameterized over a concrete program for the application and a CRDT semantics.
   b) We developed a formalized theory for combining CRDT semantics.
   c) We show that it is sufficient to check correctness in a simpler semantics, the so-called single-invocation semantics.
   d) From the single-invocation semantics we derive proof rules for deductive program verification. The soundness of the proof rules is certified using Isabelle.
3. We developed a tool called Repliss, which uses our proof technique to partially automate the process of verifying highly available applications. This includes implementations of automated testing procedures and a symbolic execution engine for verification.
4. We evaluated our technique and tool on several case studies.

1.2. Outline

We begin with a high-level introduction to our approach in Chapter 2, in which we show how the Repliss tool can be used from a user's perspective for developing and verifying a highly available application. In Chapter 3 we present background information, including current state of the art for building highly available applications, related work in verification of such applications and on verification tools in general, and a short overview over the Isabelle/HOL notations we use in the remainder of the thesis.

We then present our Isabelle formalization of replicated data types specifications in Chapter 4. Chapter 5 presents the core part of our proof technique and its soundness proof starting from an operational small-step semantics. In Chapter 6, we then further develop this technique towards the proof automation we want to achieve with the Repliss tool. To this end, we present proof rules tailored towards symbolic execution and prove their correctness.

Finally, Chapter 7 presents the design and implementation of the Repliss tool and Chapter 8 presents case studies modelled with our framework in Isabelle and with the Repliss tool.

Chapter 2

The Repliss Approach

In this section we present our approach for building verified highly available applications. Obviously, considering distributed applications in general is hard, so we restrict our scope to an application architecture for which we can develop specialized techniques.

As we want to focus on the aspects of weak consistency, we will try to contain all consistency aspects within a data layer of the application, which handles the updating and retrieving of all persistent data. This ensures that we do not have to verify the user interface and other code that is unrelated to verifying the consistency aspects of the application. Of course, the UI is not negligible when talking about consistency aspects. For example a web form that initially loads some data, which then can be edited and submitted is often found in web applications. In a strongly consistent system, editing conflicts can be resolved when submitting the form, for example by comparing versions of the underlying data and rejecting a submission if the underlying data has changed. In a weakly consistent system this is not possible and so the UI is responsible for recording more information on how the data was edited so that we get meaningful change-sets on the data that can be applied asynchronously. Also, the UI might have to display additional information about conflicting changes with mechanisms to resolve them in the UI. While these UI changes are challenging, the data layer API can be designed in a way so that we can still keep all consistency aspects contained.

With this first decision, we are left with the task of getting the data layer right. However, this is also a challenging task and there already are many databases that solve the problem of data synchronization, e.g. Antidote[1], Cassandra[2], or Redis Enterprise[3]. We want to leave generic mechanisms for synchronization and basic consistency guarantees to the database, such that we can focus on the specifics of the application. We assume that the database provides us with a certain consistency level and a set of replicated datatypes with builtin strategies for handling concurrent updates, so called

[1] https://www.antidotedb.eu/
[2] https://cassandra.apache.org/
[3] https://redislabs.com/redis-enterprise/

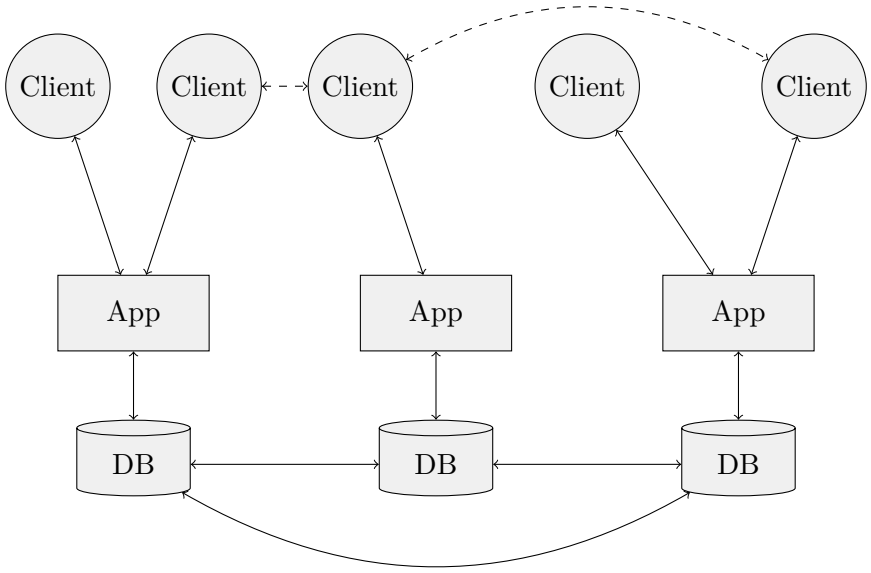


Figure 2.1.: *Visualization of architectural assumptions.*

CRDTs [Sha+11b]. Since all synchronization aspects are delegated to the database, this implies that the application itself should be stateless. Only the database should store persistent state and only the database should be used for synchronization between different invocations by possibly distinct clients of the API.

Overall, this leaves us with a high level application architecture as depicted in Figure 2.1. In the following we only consider the data layer, when we talk about applications. On top of the applications, we have clients that invoke the API of the application. We also consider the UI and other code unrelated to data management to be part of the client.

The application itself is co-located with a database replica, which allows fast local data access. The arrows in Figure 2.1 represent communication. We can see, that the different application instances can only communicate through the database. Clients are out of our control, but we have to account for the possibility that clients communicate, which we denote with dashed lines. For example, one person could send a link to a page within the application to another person via Email.

The architectural design introduced above, enabled us to develop a specialized proof technique and the accompanying Repliss tool. In the following case study, we show how Repliss can be employed in developing an application with verified correctness.

```
1  crdt chat: Set_aw[MessageId]
2  crdt message: Map_uw[MessageId, {
3      author: Register[UserId],
4      content: MultiValueRegister[String]}]
5
6  def sendMessage(from: UserId, text: String): MessageId {
7      var m: MessageId
8      atomic
9          m = new MessageId
10         call message(NestedOp(m, author(Assign(from))))
11         call message(NestedOp(m, content(Assign(text))))
12         call chat(Add(m))
13     return m
14
15 def editMessage(id: MessageId, newContent: String)
16     atomic
17         if messageQry(ContainsKey(id))
18             call message(NestedOp(id, content(Assign(newContent))))
19
20 def deleteMessage(message_id: MessageId)
21     atomic
22         if messageQry(ContainsKey(message_id))
23             call chat(Remove(message_id))
24             call message(DeleteKey(message_id))
25
26 def getMessage(m: MessageId): getMessageResult
27     atomic
28         if messageQry(ContainsKey(m))
29             return found(
30                 messageQry(NestedQuery(m, authorQry(ReadRegister))),
31                 messageQry(NestedQuery(m, contentQry(ReadFirst))))
32         else
33             return notFound()
```

Figure 2.2.: *Model of the Chat application in Repliss containing the original bug.*

2.1. Case Study

To illustrate our approach, we use an example of a highly available chat application. This example is inspired by an experience report from Discord [Vis], who migrated their chat service from a single centralized database to the replicated and weakly consistent database Cassandra [LM10]. Although the code had been well tested prior to deployment, when the new solution was first used in production, some messages ended up with missing metadata, e.g., the author field of a chat entry was empty. We now show how such a problem could have been ruled out by checking and statically verifying the application with Repliss before deployment.

The program of the chat app in Repliss is shown in Figure 2.2. In general, a Repliss program consists of three parts: (1) a data model (lines 1-4 in the example), (2) a list of procedures implementing the application (starting from line 6), and (3) some invariants specifying properties of the application (not shown in Figure 2.2, presented below).

2.1.1. Data modelling with CRDTs

In a highly available application, it is inevitable that database updates occur concurrently in different data centers without synchronization. Some updates are inherently independent of each other, for example, if they address different parts of the database state. For other cases, the application must be designed to handle concurrent updates in a meaningful way without loosing data. To achieve this, we use conflict-free replicated data types (CRDTs [Sha+11a; Sha+11b]) for modeling the data. CRDTs are abstract data types with a builtin strategy for asynchronously resolving concurrent updates. The use of CRDTs does not restrict the applicability of our technique, since typical synchronization free databases can be described in terms of CRDTs. For example many databases use a last-writer-wins strategy, which is just a special kind of CRDT.

Besides choosing the correct data types for the application data (e.g. maps, sets, lists, registers, …), programmers also have to decide how concurrent updates should be resolved. For example, the `Set` datatype comes in two variants that can be distinguished in the way how concurrent `add-` and `remove`-operations of the same element are handled. The add-wins variant, which we denote with suffix `_aw` (cf. Fig. 2.2), prefers the effect of the add-operation. More precisely, a `remove`-operation only affects the `add`-operations that have happened before the `remove` and not the concurrent `add`s. Another variant of the `Set` datatype is called remove-wins (suffix `_rw`). Here, the strategy is reversed and an `add`-operation only overwrites `remove`-operations that happened before it. Similar to sets, the `Map`-datatype also comes in a variant where updates win over concurrent deletions of entries (suffix `_uw`), and a variant where delete-operations win (suffix `_dw`). Map CRDTs can be recursive such that the embedded values are again CRDTs, which can only be updated and queried by going through the `Map` interface. For simple, atomic values, `Register` CRDTs can be used. The `Register` simply resolves concurrent updates arbitrarily (e.g. by timestamp) and returns one of the latest written values. In contrast, the `MultiValueRegister` stores the set of all latest concurrently written values (which is a singleton set if there are no concurrent assignments).

For the data model of our chat application (lines 1-4), we use an add-wins set of message identifiers named `chat` to store the set of all messages. For storing the individual messages, we use a map with the update-wins semantics. The keys of the map are message identifiers and the values are again CRDTs combining a register for the author and a multi-value register for the content.

2.1.2. Implementation Language

In Repliss, the interface and behavior of an application is realized by a set of procedures. The procedures are implemented in a simple imperative programming language. The procedures for the chat application are shown in Figure 2.2 starting from line 6. Besides the usual language constructs, like if-statements, variables, or return-statements, the language contains some constructs specific for database interaction code.

A procedure can generate new unique identifiers using the **new** keyword. To interact with the database, a call-statement is used, which starts with the keyword **call** followed by an update operation. The update operation typically is expressed using datatype constructors for the operations. For example, operations to the CRDT named `chat` are constructed with a constructor of the same name, which takes the nested update operation as a parameter. Database reads do not require a call-statement. Queries use the suffix `Qry` to be distinguishable from update operations. Several database operations can be bundled in a transaction by using an **atomic** block. A transaction guarantees that other clients interacting with the database can always see either all operations in the transaction or none of them. It should be noted however, that transactions do not provide any total order guarantees, so two transactions can be concurrent and not aware of each other.

2.1.3. Specification Language

The desired properties of the application can be specified by invariants. In Repliss, an invariant is a logical condition that must be true at any point during the execution, however invariants cannot observe changes from uncommitted transactions. Repliss supports automated testing to quickly check an invariant and verification to prove that an invariant is never violated.

To illustrate the process of using Repliss to verify an application, we consider a typical invariant of the Chat application. We require that if there is an invocation `g` of procedure `getMessage` for a message identifier `m` in our execution history that returns a message (`auth, cont`), then there has to be also an invocation of `sendMessage` with the same author `auth`:

Invariant 1

```
forall g: InvocationId, m: MessageId, author: UserId, content: String ::
    g.info == getMessage(m)
  && g.result == getMessage_res(found(author, content))
  ==> (exists s: InvocationId, content2: String ::
         s.info == sendMessage(author, content2))
```

This invariant demonstrates an essential feature of Repliss, which we call *history invariants*. The history comprises procedure invocations, database calls, and the relation between them. In the above invariant, we only address procedure invocations. The type `InvocationId` identifies a procedure invocation and ranges over all procedure invocations issued in the current execution. With the expression `g.info` we obtain the invocation information for invocation `g`, which is the name of the invoked procedure and the value of the arguments. Correspondingly, `g.result` refers the result of the procedure invocation.

Making the history available in invariants, allows us to express the effects of procedures (like `sendMessage` here) based on their influence on other procedures. This lets us avoid a state based description, where the effect of procedures would be described by changes on a global state. That approach would be difficult in our setting, since there is no global state as in a centralized system, where everyone sees the same version of the state.

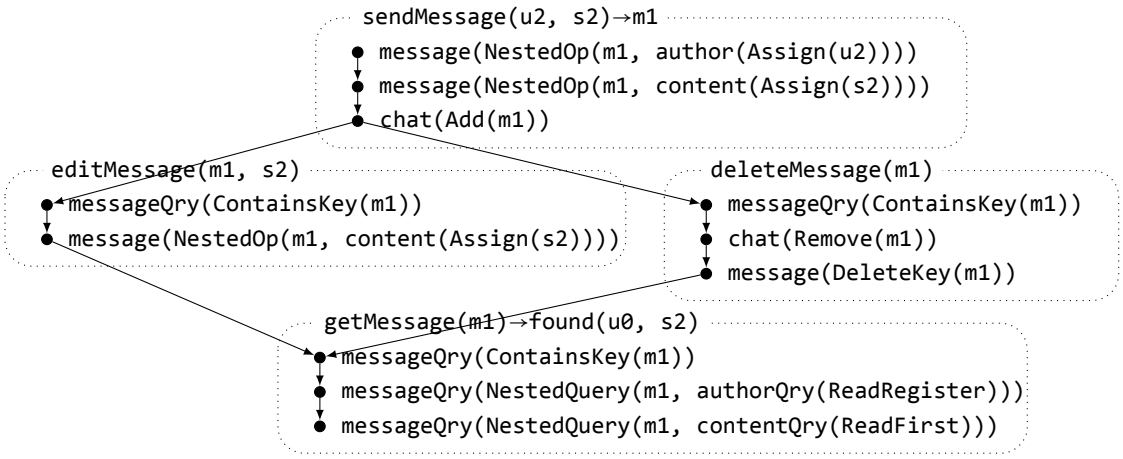


Figure 2.3.: *Counter example produced by Repliss showing an execution with concurrent deletion and updating of the same message.*

2.1.4. Tool Support

The Repliss Tool includes an automatic testing method for finding bugs and a verifier to prove the absence of bugs. We now show how these can be used on the Chat application.

When we check the application property defined in Invariant 1 with Repliss, it fails to verify the `getMessage` procedure and produces the execution in Figure 2.3 as a counter example. The example is generated by the automated testing method and shows a scenario where a user edits a message while it is concurrently deleted. When the message is read afterwards, the corresponding map-entry exists since there is a map-update that is not followed by a delete operation. However, the concurrent delete operation has removed the value from the register holding the author's name with the result that the author name is set to the default value of the register (`u0`). Thus, `getMessage` reads `u0` – a user that never sent any message. This corresponds to the bug described in Discords experience report [Vis], where the same scenario would lead to the value *null* for the author field[4].

[4]The Repliss language does not include a *null* value, so the automated testing tool will find another value. The CRDT specification would allow any value to be returned by the read operation in the given case.

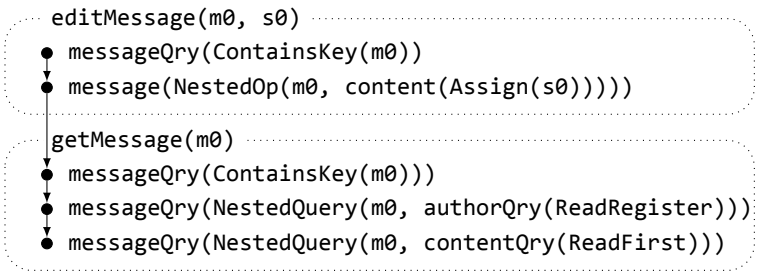


Figure 2.4.: *Verification counter example produced by Repliss (version 1).*

Once the problem is identified, we can apply a simple fix by changing the data type of `message` from `Map_uw` to `Map_dw`, so that the CRDT resolves concurrent update- and delete-operations on the map by letting the delete overwrite all concurrent updates.

With this change, Repliss is still unable to verify the correctness of the application, but the automatic testing component of Repliss no longer finds a counter example. However, the verifier of Repliss produces the counter example shown in Figure 2.4. Unlike the example produced via testing, this time we do not get a complete execution starting from the initial state. The verifier just considers one invocation (in this case `getMessage`) and the pre-state is constructed such that it satisfies the invariant, which obviously is the case since there is no invocation of `getMessage` in the pre-state. However, in the post-state, after executing `getMessage`, the invariant is violated. Therefore, we need to exclude situations like this with additional invariants. As experience shows, finding the right invariants is often the most challenging part when verifying applications. Repliss assists users in this task by visualizing a counter model for the verification condition.

The important aspect to prove our original invariant is that there cannot be an update of a message's content field without a prior assignment to the author field. Note that this is a history property as it relates different data base calls in the execution history of the system. We express this with the following invariant:

Figure 2.5.: *Verification counter example produced by Repliss (version 2).*

Invariant 2

```
forall c1: CallId, m: MessageId, s: String ::
    c1.op == Op(message(NestedOp(m, content(Assign(s)))))
    ==> (exists c2: CallId, u: UserId ::
            c2.op == Op(message(NestedOp(m, author(Assign(u)))))
            && c2 happened before c1)
```

We refer to database calls using the type `CallId`. Note that we use the word calls when referring to the database and the word invocations at the level of procedures.

After strengthening the specification with this invariant, Repliss produces the counter example in Figure 2.5, showing us that our invariant is still not strong enough. In this example, we see that the message has been deleted, but a subsequent invocation of `editMessage` adds data to the message again. Therefore, the `message_exists` check in `getMessage` returns true, but the author information is still deleted and therefore reading the author value may return an arbitrary value.

In a real execution, the check for `message_exists` in `editMessage` would have returned false, preventing the update after delete. However, Repliss does not perform this inductive reasoning and therefore we need to express it as an invariant. With the following invariant, we state that there can be no update-operations on a message after it has been deleted:

Invariant 3

```
!(exists write: CallId, delete: CallId, m: MessageId, upd ::
       write.op == Op(message(NestedOp(m, upd)))
    && delete.op == Op(message(DeleteKey(m)))
    && delete happened before write)
```

With this addition, Repliss can automatically verify the correctness of the application in 1 minute and 36 seconds. The Chat example is an instance of a typical architecture, where all communication between clients is handled via a database. This architectural restriction together with a careful choice concerning the expressiveness of invariants enables us to reduce the verification problem to a sequential one and thereby facilitates the automation described above.

In the remainder of this thesis, we will describe the theory behind the Repliss tool, which we introduced here based on an example. We will precisely define the semantics of the system, the proof technique, and we will show that our technique is sound.

Chapter 3

Background: Distributed Information Systems & Verification

This thesis combines two general fields of research: distributed systems and formal verification. As discussed in Chapter 2, our work focuses on a certain kind of distributed system. In Section 3.1 we try to illustrate, where this subset fits into the bigger picture of related systems.

From the point of view of formal verification, there also is a tremendous amount of existing approaches. Our work can be put into the general area of deductive verification of concurrent software, for which we give a short overview in Section 3.2. In Section 3.3 we then give an overview over verification techniques in the domain of highly available applications.

Finally, we give a short introduction to the Isabelle/HOL interactive theorem prover and its notations in Section 3.4, as we make heavy use of it in the following chapters.

3.1. Distributed Information Systems

In this thesis, we consider information systems that use replication. This means that the data of the application exists as copies at several locations. Replication can be used to improve availability, fault tolerance, or performance.

Many systems used in production today are built using ad-hoc methods to implement the replication and scaling. One well known example is Facebook, which published information about its database backend called TAO [Bro+13]. The system uses sharding to distribute the data to multiple MySQL databases. To scale geographically, the system uses a master-slave replication where updates can only occur on the slave, but reads can also be served by slaves. Additionally, there are multiple levels of caches for speeding up read-operations. Since caches and slaves are updated asynchronously after an update, there can be stale reads though. One special case is reading a value after a write from the same node.

Fault tolerance is achieved by trying to detect failures. If a failure of a master database node is detected, one of its slaves becomes the new master. However, such a case can lead to split brain scenarios and inconsistencies. The TAO papers mentions Googles Spanner [Cor+12] as an alternative with consistency guarantees, but state that it could not handle the necessary number of requests. Moreover, Spanner needs special hardware support (GPS and atomic clocks) to guarantee some upper bounds on clock uncertainty. Other strongly consistent alternatives that support updates at multiple replicas use consensus protocols like Paxos [Lam98] or Raft [OO14] and suffer from high latency and reduced availability when network partitions occur.

The applications we consider in this thesis use a different kind of database systems that lie between the two extremes introduced above. In these databases, updates can be performed on all replicas like in Spanner but unlike strongly consistent databases, updates do not need to wait for synchronization. In contrast to the two types of database systems above, this implies that there can be concurrent updates that are not aware of each other and that must be merged asynchronously.

There are several options to implement this merging. The easiest solution is to use timestamps and to just keep the latest version. For example, this is the default option for Amazon S3 [Ser20a]. To prevent data-loss, one can also keep versions of earlier updates. This option is also available on Amazon S3 [Ser20b]. Other options are specific to certain data structures, like operational transformation [EG89] or Conflict-Free Replicated Data Types (CRDTs) [Sha+11b]. From an application's perspective, all of these techniques can be seen as instances of replicated data types, which we handle in detail in Chapter 4.

Another aspect how highly available database distinguish themselves, are the consistency levels they support. There are a few orthogonal aspects in consistency models. A good overview over non-transactional models is given by Viotti and Vukolic [VV16]. We focus here on session models and causal models, as these are the models that can be implemented in a highly available way, i.e. without updates waiting for synchronization. In fact, Attiya, Ellen, and Morrison showed that no consistency model stronger than observable causal consistency can be implemented in such a way [AEM17].

One aspect of consistency models are the provided session guarantees [Ter+94]. These are guarantees related to the session order, which is the order in which a single client or process performs operations. In our case, a session is given by a procedure invocation. We assume a consistency model satisfying all session guarantees. However, this is not an essential choice for our technique and it could be changed to weaker session models.

Another aspect of consistency models are restrictions on the order of operations. In this thesis, we use the causal consistency model [Llo+11]. This model provides the following guarantee: If the effect of an operation x is visible and the effect of an operation y was visible when executing x, then the effect of y must also be visible. In other words, the happens-before relation must be transitive. Our restriction to causal consistency model is not necessary for

all parts of our technique. Stronger models can still be expressed by order constraints in the invariants, but handling mutual exclusion efficiently would require different proof techniques. Our techniques could be adapted to weaker consistency models. However, in weaker models even fewer properties would hold and while weaker models can provide better performance, there would be no benefit in terms of availability. So we did not consider it worth the effort to parameterize our whole system model and technique with a consistency model.

Finally, there is the aspect of transactions in consistency models. Transactions are a mechanism to group a set of operations and provide certain guarantees about the visibility of these operations. Typically, they guarantee atomicity, which means that either all or none of the operations in a transaction are visible to other observers. Transactions also provide some form of isolation from concurrent operations, with different levels of strictness and ordering guarantees [CBG15]. In our work, we consider transactions that work on a fixed snapshot [ASS13] and thus cannot observe concurrent updates. This choice still allows modelling of systems without transactional support, by simply using one transaction per operation.

3.2. Specification and Verification

When we talk about software verification, we mean the process of proving that an implementation or model of the software satisfies a specification. This adds some redundancy to the software development process, as the behavior is described at least twice. Often, it is an executable implementation and a specification describing parts of the intended behavior at a higher level. Software quality is increased because the behavior of the system is described from different perspectives, and because the high level specification is often easier to understand and relate to the informal specification of the software.

There are two main approaches to software verification: Fully automated methods (model checking) and deductive techniques [Pel01]. Model checking is restricted to certain kind of systems, for which the state space can be explored automatically. This often means that the system must be representable using a finite state system or one, where certain abstractions can be used to make the state space finite. In contrast to this, deductive verification derives logical formulas, so-called verification conditions, from the program code. If these formulas can be proven to be valid in general (or unsatisfiable, depending on the formalization of the problem), then it is implied that the program satisfies this specification. This approach is more flexible in the kind of systems and the power of specifications it can support. However, deductive verification typically requires some user interaction and cannot be fully automated for all cases. In this thesis we follow the deductive approach to benefit from this flexibility. Like others, we aim to automate parts of the process by relying on SMT solvers [MB08; Bar+11].

Concurrency in Deductive Program Verification

A central challenge of our work is the handling of concurrency. Distributed systems are inherently concurrent, since they involve processes running on multiple machines, and we do not want to restrict the system to only have one machine executing code at a time. Concurrency is challenging in normal software development as well as in software verification because the interleaving of concurrent processes increase the number of possible executions that must be considered.

Early work by Ashcroft [Ash75] uses invariants to reason about all possible interleavings. We tried this simple method to verify a small example of a user database (see Section 8.2) but failed to do so because the complexity of invariants increased with every additional statement in a process.

More recent techniques for verifying concurrent systems exploit that concurrent programs are often programmed to behave similar to sequential programs. In particular, when two processes share no mutable memory, they can be reordered to an equivalent sequential execution. When they need to communicate, mechanism like locks can be used to guarantee exclusive access to one process. Probably the first proof system for exploiting noninterference in concurrent processes was developed by Owicki and Gries [OG76]. This method was later defined to be easier to compose in rely-guarantee techniques [Jon83]. Another central idea is the concept of ownership. One instance of a logic with ownership is separation logic [Rey02], which can also be combined with rely-guarantee reasoning [Vaf08]. There are several tools that use ownership to reason about concurrency, for example Spec# [Jac+08], Chalice [LMS09], Verifast [Jac+11], or Viper [MSS16].

We looked at ways to transfer the ideas from these techniques to our setting. However, one essential difference is that we do not have specifications, which relate the concurrent executions to some sequential execution. For example, when concurrent edits of the same register should result in storing all the latest concurrent values (see multi-value register in Section 4.1), then no sequential execution could explain such an outcome.

This is different, even from work on weak memory models, for which the specification of verified software typically includes a relation to a sequential model, as in causal linearizability [Doh+18].

Specification of Concurrent Behavior

While in sequential applications, the effect of a piece of code can be described by a simple predicate relating the pre- and post-state, the situation is more complex for concurrent code. This is because concurrently executed code can change the same state. Therefore, it is necessary to isolate the changes from other code. In rely-guarantee [Jon83] this aspect is handled by using 2-state predicates. A postcondition describes the overall effect of the procedure on the state, while the guarantee-condition describes the possible effects of the individual atomic actions in the code. We use a similar technique of relating two states in our technique (see Section 5.2). However, for our specifications it

is not sufficient to be able to relate two global states, since there is no notion of a global state.

So, instead of focussing on states, we follow a specification technique by Gotsman and Yang [Bur+14; GY15a], where the focus is on events and relations between events. Event histories are also used for specifying actor systems in ABS [DO14] or in temporal logics like in TLA$^+$ [Lam02]. However, these do not use a partial happens-before relation which is important for our applications.

3.3. Related Work

The challenge of weak consistency in verification is well known and has been approached with a number of different techniques. Weak memory models have been studied in depth in the context of concurrent programming for multi-core machines [DD15]. However, the techniques in this area usually target linearizability as a correctness criterion and employ hardware-supported synchronization mechanisms such as memory fences or CAS-operations. In distributed systems, it is neither feasible to consider linearizability as consistency notion nor to implement the same concurrency control mechanisms as in weak memory system. This precludes the direct applicability of these techniques to our scenario. In the following, we therefore focus on related work that shares our application domain.

Composite Replicated Data Types [GY15b] allow to compose basic data types into application-specific data representations that are synchronized atomically. Their area of application is similar to our setting, though our approach is more widely applicable as we model procedures involving several transactions on arbitrary combinations of objects. More importantly, their approach is axiomatic and based on a denotational semantics, which is more difficult to adapt in a tool implementing the technique.

CISE [Got+16; Naj+16] is a tool, which can automatically determine the procedures in an application, which require stronger consistency guarantees for correctness. This line of work focuses on combining weak consistency with strong synchronization for some operations, whereas our work only considers weak consistency. CISE does not consider features like transactions or replicated data types directly. Instead, application procedures are assumed to have a single atomic effect which is applied on every replica asynchronously. This is similar to the implementation technique of operation-based CRDTs, where effects have to be commutative to ensure convergence. Soteria [NPS19] is a similar tool which is based on state-based implementations of CRDTs instead. In contrast, our model handles data types as components with a high-level (axiomatic) specification and not their concrete implementation.

QUELEA [SKJ15] is another tool supporting the development of applications on top of weakly consistent databases. Unlike our approach and the previously discussed approaches, the specifications in QUELEA are not given as invariants. Instead, the user specifies constraints on the order between operations and the tool automatically chooses the necessary consistency level.

Carol [Lew+19] is a programming language which automatically manages consistency levels based on consistency guards. For example a consistency guard on a read operation can require that no concurrent operation decreases the value of a counter below the value that was read. In this case, the system would prevent all decrement operations until the operations depending on the read are completed. Thus, similar to QUELEA, the tool does not work with specifications given as invariants, but with constraints that the programmer must choose correctly.

Q9 [Kak+18] is a symbolic execution engine for finding bugs in programs written on top of weakly consistent databases. The tool only supports bounded verification, where the number of concurrent effects is limited, so unlike Repliss, it cannot be used to prove the absence of errors in the general case. Weak consistency is modeled using commutative effects, which works well for symbolic execution, but is less suitable when working with invariants as we do.

Chapar [LBC16] is a framework for verifying causally consistent, replicated databases and applications employing such databases. The development is formally verified using Coq and the goal of verifying application is similar to ours. Their approach is different, though. They have implemented a model checker for applications, thus providing automation. However, the kind of applications which can be analyzed is restricted, since the model checker can only check all possible reorderings of one concrete execution where all parameters have fixed values.

None of the work discussed so far handles the integration of transactions into a technique to reason about programs. This aspect has been tackled in work in different contexts, for example in a program logic for handling Java Card's transaction mechanism [BM03]. Transactions in Java Card provide atomicity, but do not handle concurrency. We are not aware of other work integrating weakly consistent transactions into a deductive verification technique.

3.4. Isabelle/HOL

Isabelle/HOL [NPW02] is an interactive theorem prover, which we use in this thesis to formalize the semantics and for writing machine checked proofs. In this section, we give a short overview over the Isabelle notation we use in the following chapters.

At its core, Isabelle's syntax for terms and types is similar to languages like ML [MTH90]. Function application is written without parenthesis, e.g. $f\ x\ y$. Function abstractions are written as $\lambda x.\ e$. Besides these basic constructs, Isabelle supports user defined mixfix syntax [Fut+85] with custom binders, which is used by Isabelle's standard library to define some basic language constructs. This includes conditional expressions ($\mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2$), let bindings ($\mathit{let}\ x = e_1\ \mathit{in}\ e_2$) and pattern matching ($\mathit{case}\ e\ \mathit{of}\ p_1 \Rightarrow e_1 \ldots \mid p_n \Rightarrow e_n$).

Type annotations are given with the syntax $e :: T$, stating that expression e has type T. If no type annotations are given, types are inferred.

Type	Values	Description
bool	*True, False*	Boolean values
nat	*0, 1, 2*	Natural numbers
int	*-1, 0, 42*	Integers
'a ⇒ 'b	$\lambda x.\ x + 1$	Function from *'a* to *'b*
'a × 'b	*(x, y)*	Pairs
'a option	*None, Some 5*	Optional values
'a set	*{1, 2, 3}*	Sets
'a ⇀ 'b	$[x \mapsto a, y \mapsto b]$	Maps (synonym for *'a ⇒ 'b option*)
'a list	*[1, 2, 3]*	Lists
'a rel	*{(1, 2), (2, 3)}*	Relations (syn. for *('a × 'a) set*)

Figure 3.1.: *Basic types defined in the Isabelle standard library.*

3.4.1. Standard Types and Functions

Isabelle's has a Hindley-Milner style type system [Hin69; Mil78] with type classes [WB89] and records with row polymorphism [Wan91].

The basic types of Isabelle are *bool* for Boolean values and $T_1 \Rightarrow T_2$ for functions. Type variables are prefixed with an apostrophe as in *'a*. Parameterized types are written with the type parameters before the type name as in *'a list*.

The Isabelle standard library provides some commonly used types. In Figure 3.1 we give an overview over the types we use in this thesis. Below we show the common operations on these types:

Functions The syntax $f(a := b)$ denotes function updates. It is equivalent to $(\lambda x.\ \mathit{if}\ x = a\ \mathit{then}\ b\ \mathit{else}\ f\ x)$.

Booleans The type *bool* supports the standard operations conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), equivalence ($\leftrightarrow$), and negation ($\neg$). We also have existential ($\exists$) and universal ($\forall$) quantifiers. The notation uses a dot to separate variables from the quantified formulas as in $\forall x.\ P\ x$ for one variable x or $\forall x\ y.\ P\ x\ y$ for two variables. Note that the quantification implicitly ranges over all values belonging to the type of the variable, which is usually not given explicitly.

Pairs Tuples are constructed with the usual syntax (a, b, c). All tuples are represented by pairs. The expression (a, b, c) is equivalent to $(a, (b, c))$. The functions *fst* and *snd* are used to extract the first or second component of a pair.

Optional Values The two constructors for optional values are *None* and *Some*. We use the shorthand $x \triangleq y$ for $x = \mathit{Some}\ y$.

Sets Isabelle supports the standard set operators: inclusion of elements ($\in$), subsets ($\subseteq$), proper subsets ($\subset$), union ($\cup$), intersection ($\cap$), complements ($-S$),

and cartesian products ($\times$).

Quantification can be limited to the elements of a set. The expression $\forall x \in S.\ P\ x$ is equivalent to $\forall x.\ x \in S \rightarrow P\ x$ and $\exists x \in S.\ P\ x$ is equivalent to $\exists x.\ x \in S \wedge P\ x$. This syntax can be used with patterns for pairs as in $\forall (x, y) \in S.\ x < y$.

Maps Maps are functions to an option type. Since they are functions, we can use the same update syntax. In addition, we use $f(x \mapsto y)$ as a shorthand for $f(x := \mathit{Some}\ y)$.

The function *dom* returns the domain of a map m, i.e. all the values x, such that $m\ x \neq \mathit{None}$. Similarly, *range* returns the range of a map m, i.e. all values y, such that there is an x with $m\ x \triangleq y$.

A map m can be restricted to a given set S with the expression $m{\restriction_S}$, which is equivalent to $(\lambda x.\ \mathit{if}\ x \in S\ \mathit{then}\ \mathit{Some}\ (m\ x)\ \mathit{else}\ \mathit{None})$.

Relations A relation is a set of pairs and as such supports all set operators. Isabelle supports predicates *trans* to check if a relation is transitive, *refl* for reflexivity, *acyclic* for the absence of cycles, and *antisym* for checking whether a relation is antisymmetric.

A relation R can be restricted to a set S with the syntax $R\ |_r\ S$, which is equivalent to $R \cap (S \times S)$.

Lists Lists are finite datatypes with constructors *Nil* (also written $[]$) and *Cons* (also written $x\#xs$ or $x \cdot xs$). The operator @ appends to lists, the function *length* returns the length of the list and the function *set* converts a list to a set. The expression $xs!i$ (also written as $xs_{[i]}$) returns the element at position i in the list, where the first element is at index 0. The function *distinct* checks whether all elements in a list are distinct.

3.4.2. Type Definitions

There are three basic ways to define new types, which we use in this thesis: type synonyms, data types, and records.

Type synonyms allow to define a new name for an existing type. For example the following type synonym is the definition for the type *rel*:

type-synonym $'a\ \mathit{rel} = ('a \times 'a)\ \mathit{set}$

Datatype definitions consist of one or more constructors. The constructors can have parameters, which can optionally be named to generate the respective selector functions. As an example consider the definition for the *option* type:

datatype $'a\ \mathit{option} =$
  None
| $\mathit{Some}\ (\mathit{the}\colon 'a)$

Records consist of one or more fields. The field names are also selector functions for the field. The following example shows two record definitions, where the second record extends the first:

record *product* =
  name :: *string*
  price :: *nat*

record *food* = *product* +
  calories :: *nat*

A record definition defines two types: The record type itself (e.g. *product*) and a type for the record-scheme (e.g. *product-scheme*) which is similar to the record type, but contains an additional slot of a type *'a* which stands for possible extensions to the record. A function that takes a *product-scheme* parameter can be used with both type *product* and type *food*.

A record is constructed using banana brackets (for example $b \equiv (\!|name = ''Banana'', price = 55, calories = 89|\!)$). Fields are updated with a syntax similar to function updates as in $b' \equiv b(\!|price := 53|\!)$.

3.4.3. Type Classes

Isabelle supports type classes [WB89]. Polymorphic definitions can restrict type parameters to types that belong to certain type classes. A type class can require that a type provides specific constants and functions. Unlike programming languages like Haskell a type class can also require that the type satisfies certain properties. Type classes can also extend other type classes, inheriting the respective constraints.

The following example defines a class named *valueType*, which extends the *countable* and *default* type classes. It requires that there is a function *uniqueIds*, which assigns a set of unique identifiers to all elements belonging to the class. Furthermore, it restricts this function by stating that the default value of the type must contain no unique identifiers.

class *valueType* = *countable* + *default* +
  fixes *uniqueIds* :: *'a* ⇒ *uniqueId set*
  assumes *default-none*: *uniqueIds default* = {}

Type classes can be used wherever a type variable can be used. To restrict a type variable, it can be annotated with a type class as in *'a::valueType*. Several type class restrictions are notated using curly braces (e.g. *'a::{countable, default}*).

3.4.4. Definitions

In this thesis, we use three different kinds of definitions available in Isabelle. First there are simple constant definitions. We denote such definitions with the $(\equiv)$ symbol, as in $f\ x \equiv x + 1$.

Second, we use recursive function definitions. There are marked with the keyword *fun* and can consist of one or more cases (separated by a vertical bar). The implementation of the cases can include recursive function calls. Each recursive function definition must be proven to be well-defined, which Isabelle can often do automatically.

The other kind of definition we use, are inductive predicates. These consist of one or more cases, which are implications of the form $A_1 \wedge \ldots A_n \Longrightarrow P$. The semantics of inductive predicates is the least fixedpoint for the predicate that is induced by the implications. Isabelle enforces that the implications are monotone which guarantees the existence of the least fixed point.

3.4.5. Theorems

A theorem is marked by the keywords *lemma* or *theorem*. For theorems, there are variants for some logical operators which are relevant for how theorems can be used in proofs. We call these operators of the meta logic. We use $\bigwedge$ instead of $\forall$ for universal quantification and $\Longrightarrow$ instead of $\rightarrow$ for implication. Free variables in a theorem are implicitly quantified universally. Moreover, double square brackets with semicolons can be used to group several assumptions in implications: The theorem $[\![A; B]\!] \Longrightarrow C$ is equivalent to $A \Longrightarrow B \Longrightarrow C$. On the top-level the keywords *fixes*, *assumes*, *shows*, with the conjunction *and* can be used instead of the meta operators. This syntax also allows naming the assumptions of the theorem. For example, the theorem $\bigwedge x :: int.\ [\![A\ x; B\ x]\!] \Longrightarrow C\ x$ can be written as:

lemma *example*:
  fixes $x :: int$
  assumes *foo*: $A\ x$
    and *bar*: $B\ x$
  shows $C\ x$

Each Lemma is followed by a proof using the methods provided by Isabelle. However, this document does not contain the Isabelle proofs and just gives the outlines of the formal proofs.

Chapter 4

Replicated Data Types

The use of replicated data types (CRDTs) [Sha+11b; Sha+11a; Pre18] is a major aspect of our technique for developing and verifying highly available applications. The main idea of CRDTs is that they are data types with a builtin strategy for replication. Each replica of a datatype can be deployed on a different machine and operations can be performed on each replica without waiting for synchronization with other replicas. Communication between replicas can be done completely asynchronously and concurrent updates are resolved automatically according to some well-defined strategy. This strategy ensures that two replicas end up in the same observable state when they have synchronized the same set of updates. Because all concurrent updates must be resolved by the strategy of the CRDT, they are sometimes called *conflict-free*, which is one of the origins of the "C" in the abbreviation "CRDT". Other origins of the "C" depend on the implementation technique, of which there are two major variants:

1. State based implementations are also called **C**onvergent Replicated Data Types. The main idea of this technique is that replica states form a join-semilattice and all updates result in bigger states with respect to the semilattice. For synchronization, replicas send their state to other replicas where the join with the local state is applied. The properties of the semi-lattice ensures that replicas always end up in the same state after exchanging their states. Variants of this technique [ASB18; Ene+19] only exchange fragments of the state affected by updates.

2. Operation based implementations are also called **C**ommutative Replicated Data Types. Here, the main idea is that replicas transform each update operation into a so-called downstream operation. This downstream operation is then applied to the local replica and to all other replicas. To ensure convergence, the downstream operations must be commutative with all concurrent downstream operations.

However, our technique is not dependent on any specific implementation technique for the data types. Instead, we base our technique only the specifi-

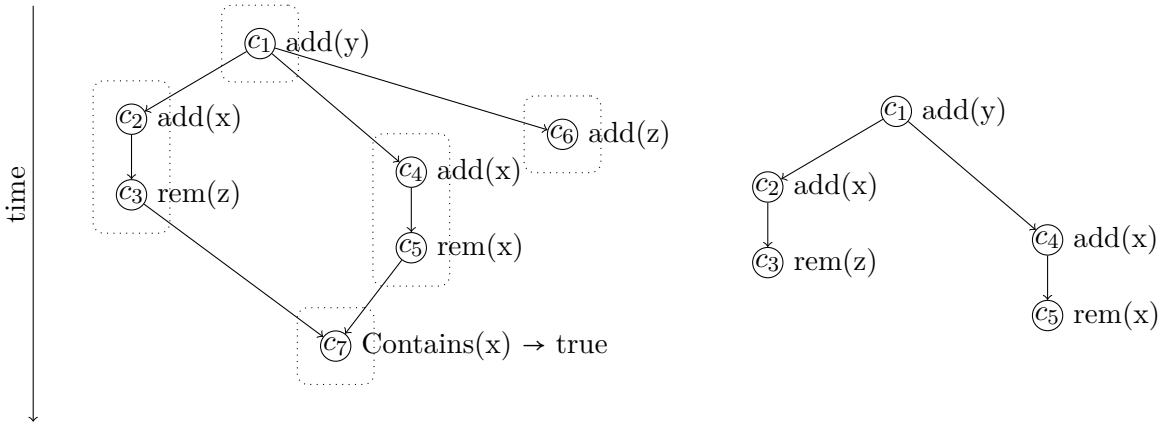


Figure 4.1.: *Illustration of an event graph (left) and the extracted operation context for query c_7 (right).*

cation of the functional behavior of CRDT based on event graphs. This specification technique was first introduced by Gotsman and Burckhardt [Bur14] and since then used in other publications [ZBP14b; Att+16; GY15a; GY15b; Bur+14]. In particular, our own work [ZBP14b] uses Isabelle/HOL to formally show the correspondence of some state-based implementations to a specification based on event-graphs.

As an example, consider Figure 4.1, which shows an event graph of an execution involving a replicated set. Each call to the database is represented by a node. Several calls can be bundled together in a transaction (boxes with dashed line). We add an edge from a call c_1 to a call c_2 if call c_1 happened before call c_2. Calls that are not reachable from each other, such as c_2 and c_5, are concurrent. The result of a read operation, like c_7 in the example, depends only on the calls that happened before. The subgraph of calls that happened before an operation is called the operation context. For query c_7, the operation context is visualized on the right part of Figure 4.1. In the operation context, details like transactions are no longer present. Given an operation context and the parameters of the database operation, the specification (*querySpec* in the formal model) of the corresponding data type yields the result of the operation. For example, the specification of a *Contains* operation on an add-wins replicated data type is defined as:

set-aw-spec (*Contains x*) *ctxt res* =
(*res* =
from-bool
(∃ *a*. *Op ctxt a* ≜ *Add x* ∧
(∄ *r*. *Op ctxt r* ≜ *Remove x* ∧ (*a*, *r*) ∈ *happensBefore ctxt*)))

Here *Op* is an abbreviation to retrieve the operation for a database call from the operation context and *from-bool* is a function to convert a boolean to the value type *'any*. Since no remove happened after the $add(x)$ in c_2, the read operation c_7 from our example returns a set containing x. Though we do not model replicas explicitly, they are represented as concurrent events in the event graph.

Composing CRDT semantics The main challenge in formalizing CRDT semantics lies in the composition of CRDTs, in particular with data types that can have arbitrary nested data types. The most common case is a map data type, where the values in the map are again replicated data types.

One option to handle nested data types is to calculate a view of the event graph that focuses on the nested operations. This leads to relatively straight forward specifications. We present a corresponding collection and framework for CRDT semantics in the first Subsection 4.1.

The downside of this approach is that it leads to complex higher-order formulas in the generated verification conditions. These formulas are hard to handle with the automatic tools of Isabelle and even more difficult to use with an off-the-shelf SMT solver, which only supports first-order formulas. In particular, when calculating the sub-context, the mapping back to the original context is lost, so it is hard to reconstruct the original calls from the calls in an embedded context.

We will therefore present a slightly different approach to composing CRDT semantics in Subsection 4.2. This approach does not explicitly calculate a sub-context. Instead, specifications now depend on additional arguments, which define which events are visible and how events have to be transformed.

4.1. Higher-Order CRDT Specifications

We now present our first formalization of CRDT specifications in Isabelle. We call the specifications in this section *higher-order*, since they implement composition of CRDTs by applying the specification of embedded CRDTs on transformed operation contexts. This technique can easily be expressed in Isabelle's higher order logic, but is difficult to translate to first order logic.

A CRDT specification is a predicate of type $'op \Rightarrow ('op, 'res)\ operationContext \Rightarrow 'res \Rightarrow bool$. The type parameter $'op$ is the type of operations and $'res$ is the type of operation results. The parameter $'res$ will always be instantiated to the value type of programs, which we refer to as $'any$ in the following chapters. The shape of an operation context is described by the following type definitions:

datatype $('op, 'any)\ call = Call\ (call\text{-}operation\colon 'op)\ (call\text{-}res\colon 'any)$

record $('op, 'any)\ operationContext =$
$calls :: callId \rightharpoonup ('op, 'any)\ call$
$happensBefore :: callId\ rel$

type-synonym $('op, 'res)\ crdtSpec = 'op \Rightarrow ('op, 'res)\ operationContext \Rightarrow 'res \Rightarrow bool$

A database *call* contains the issued operation and the returned result. Each call is identified by a *callId*. The *operationContext* record contains the mapping *calls* with the information for each call visible in the context and the *happensBefore* relation on these calls, which is a strict partial order (transitive and irreflexive).

```
datatype counterOp =
    Increment int
  | GetCount

definition
increments op ≡ case op of Increment i ⇒ i | - ⇒ 0

definition counter-spec :: (counterOp, 'a::{default,from-int}) crdtSpec where
counter-spec oper ctxt res ≡
  case oper of
    Increment i ⇒ res = default
  | GetCount ⇒ res = from-int (∑(-,c)←m calls ctxt. increments (call-operation c))
```

Figure 4.2.: *Specification of Counter CRDT*

Definition of CRDTs. To specify a new replicated datatype we have to define a new type for the operations provided by the replicated data type. This type has to provide certain operations that are required by our verification framework and for composing specifications.

To make an operation type usable with our verification framework, it must be possible to enumerate the unique identifiers used in an operation. We therefore make each operation type an instance of the *valueType* typeclass. In addition to this, for the purpose of composition we need to distinguish update operations from query operations. To this end, we introduce another type class named *crdt-op*, which extends the *valueType* class and provides a predicate *is-update* to check whether an operation is an update operation.

We now present a collection of CRDT specifications using the above technique.

Counter

A counter is an integer value that can be manipulated with an *Increment* operation. This operation can also take a negative value to decrement the counter. The *GetCount* operation returns the current count of the counter. For the specification, we simply sum the increments from all database calls in the given context.

The formalization of the specification is given in Figure 4.2. The result type *'a* of the specification needs to implement two typeclasses: The *default* typeclass is required to provide a default value for operations that do not return a result, which in case of the counter is the *Increment* operation. For specifying the *GetCount* operation, we need to convert the counter value to the value type *'a*. To this end, we use the *from-int* type class which provides the needed conversion function with the same name.

Registers

A register stores a single value of some arbitrary type. The value can be updated with the *Assign* operation and retrieved with the *Read* operation.

The formal specification is given in Figure 4.3. The returned value depends only on the latest assignments in the operation context, which are the *Assign* operations that have not yet been overridden by other *Assign* operations. The definition *latestAssignments* formalizes this selection of the relevant database calls.

We then specify two kinds of registers. In both cases the specification is parameterized by the initial value of the register, which is returned by *Read* if there are no *Assign* operations in the context.

In the first specification (*register-spec*), we define the result to be some value from the set of latest values. This means that the result is not deterministic if there is more than one latest value.

This is improved in the specification *lww-register-spec*. Here, we use the function *firstValue* to get the value from the latest assignments which has the smallest *callId*. In practice, the total order on calls is often given by a combination of a timestamp with a site identifier. For our specification, we simply use an arbitrary, but fixed total order on *callIds*[1].

Multi-value Register

The multi-value register uses the same operations as the previously described registers, but it returns a value containing all the latest values instead of a single value from this set. The formal specification for this is given in Figure 4.4. The specification requires that the result type $'a$ can represent finite sets of elements. To this end we use the type class *is-set*, which provides a predicate with the same name. The predicate *is-set* v S checks that the value v of type $'a$ is a set containing all the values in the set S.

We use a predicate here to allow more flexibility when using the specification. For example, the concrete value type could have a case for lists, but not for sets and could still implement the *is-set* type class. In that case, all orderings of the list would be valid return values as long as they contain the same elements as the latest values assigned to the register.

Flags

A flag stores a single boolean value. It provides two operations to *Enable* or *Disable* the flag. There are several semantics possible for flags. Unlike registers, which store a value transparently and thus cannot use properties of the stored values to resolve conflicts, flags can give precedence to *Enable* or to *Disable* operations. We call these *Enable*-wins (*ew*) and *Disable*-wins (*dw*) strategies. In addition to this choice, we can specify the initial value of the flag to be
True or *False*. For our specifications, we chose *False* as the initial value to be consistent with the default value for Booleans in most programming languages.

In addition, there are at least two sensible ways to give precedence to an operation, where one is stronger than the other. Since one variant gives a

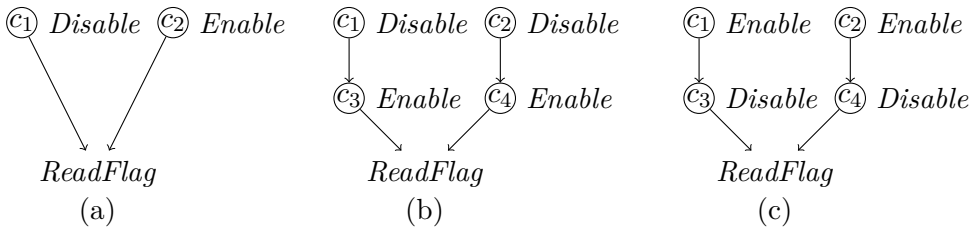
[1] Such an order always exists which is proven as a corollary of Zorn's Lemma in *HOL/Zorn.thy* of Isabelle's standard library.

```
datatype 'a registerOp =
    Assign 'a
  | Read

definition
latestAssignments-h c-ops s-happensBefore ≡
   λc. case c-ops c of
    Some (Assign v) ⇒
        if ∃ c' v'. c-ops c' ≜ Assign v' ∧ (c,c')∈s-happensBefore then None else Some v
  | - ⇒ None

definition latestAssignments :: ('a registerOp, 'r) operationContext ⇒ callId ⇀ 'a
where
latestAssignments ctxt ≡ latestAssignments-h (Op ctxt) (happensBefore ctxt)

definition
latestValues ctxt ≡ Map.ran (latestAssignments ctxt)

definition register-spec :: 'a::default ⇒ ('a registerOp, 'a) crdtSpec where
register-spec initial oper ctxt res ≡
  case oper of
    Assign x ⇒ res = default
  | Read ⇒ if latestValues ctxt = {} then res = initial else res ∈ latestValues ctxt

definition lww-register-spec :: 'a::default ⇒ ('a registerOp, 'a) crdtSpec where
lww-register-spec initial oper ctxt res ≡
  case oper of
    Assign x ⇒ res = default
| Read ⇒ res = firstValue initial (latestAssignments ctxt)
```

Figure 4.3.: *Specification of Register CRDTs*

```
definition mv-register-spec :: ('a registerOp, 'a::{default,is-set}) crdtSpec where
mv-register-spec oper ctxt res ≡
  case oper of
    Assign x ⇒ res = default
| Read ⇒ is-set res (latestValues ctxt)
```

Figure 4.4.: *Specification of Multi-Value CRDTs*



	(a)	(b)	(c)
ew	1	1	0
sew	1	1	1
dw	0	1	0
sdw	0	0	0

Figure 4.5.: *Different semantics for Flag CRDTs.*

stronger precedence to an operation, we add an *s* prefix to the stronger variant (*sew* and *sdw*). We illustrate these alternatives with the examples in Figure 4.5. In example (a) we only have two updates: a *Disable* operation concurrent with an *Enable* operation. Here, the stronger variant yields the same result as the weaker one. The difference only materializes in longer executions as in example (b) and (c).

The weaker variant is similar to registers in the sense that only the latest operations, i.e. the operations that have not been overridden by other updates, are relevant for the result. The *Disable*-wins strategy (*dw*), where the value of the register is *False* if the set of latest operations contains a *Disable* operation, and the *Enable*-wins strategy (*ew*), where the value is
True if it contains a *Enable* operation.

The reasoning behind the stronger variant is different. Here, we state that the preferred operation overrides all previous and all concurrent operations. For the strong *Disable*-wins (*sdw*) strategy, this means that an *Enable*-operation only has an effect if it was executed after all *Disable* operations in the context. So in example (b), the enabling in c_3 has no effect since it was not aware of the concurrent disabling in c_2 and c_4 has no effect because of c_1. Thus, the *sdw* semantics yields a different result from the weaker one.

The formalization of our flag specifications is given in Figure 4.6. Note that the *Disable*-wins semantics are slightly longer because we want the initial value to be *False* for all semantics. If it were not for the initial value, the enable- and disable-wins would be completely symmetric.

Of course other semantics are possible as well:

G-flag It is possible to define a flag that can only be enabled and never disabled afterwards. We call this the grow-only flag.

2P-flag As a slight extension of the previous point, we can define a flag that can be disabled only once and then never be disabled again. The value is
True if and only if there is an *Enable* operation in the context and no *Disable* operation. We call this the 2-phase-flag.

```
datatype flagOp = Enable | Disable | ReadFlag

definition
latestOps ctxt ≡
  {op | c op.
          Op ctxt c ≜ op
        ∧ is-update op
        ∧ (∄ c' op'. Op ctxt c' ≜ op'
                  ∧ is-update op'
                  ∧ (c, c')∈happensBefore ctxt)}

definition flag-dw-spec :: (flagOp, 'a::{default,from-bool}) crdtSpec where
flag-dw-spec oper ctxt res ≡
  case oper of
   ReadFlag ⇒ res = from-bool (Enable ∈ latestOps ctxt ∧ Disable ∉ latestOps ctxt)
  | - ⇒ res = default

definition flag-ew-spec :: (flagOp, 'a::{default,from-bool}) crdtSpec where
flag-ew-spec oper ctxt res ≡
  case oper of
   ReadFlag ⇒ res = from-bool (Enable ∈ latestOps ctxt)
  | - ⇒ res = default

definition flag-sdw-spec :: (flagOp, 'a::{default,from-bool}) crdtSpec where
flag-sdw-spec oper ctxt res ≡
  case oper of
   ReadFlag ⇒ res = from-bool (∃ e. Op ctxt e ≜ Enable
                ∧ (∀ d. Op ctxt d ≜ Disable ⟶ (d,e)∈happensBefore ctxt))
  | - ⇒ res = default

definition flag-sew-spec :: (flagOp, 'a::{default,from-bool}) crdtSpec where
flag-sew-spec oper ctxt res ≡
  case oper of
   ReadFlag ⇒ res = from-bool ((∃ e. Op ctxt e ≜ Enable)
                ∧ (∄ d. Op ctxt d ≜ Disable
                  ∧ (∀ e. Op ctxt e ≜ Enable ⟶ (e,d)∈happensBefore ctxt)))
  | - ⇒ res = default
```

Figure 4.6.: *Specification of Flag CRDTs*

Lww-flag We can use an arbitration order like timestamps to resolve conflicts. This is similar to the last-writer wins register. In contrast to registers it is sound to use non-unique timestamps (i.e. the arbitration order can be any acyclic ordering relation) if updates with equal timestamp (i.e. equal or incomparable w.r.t. arbitration order) are resolved deterministically, for example using a precedence on values.

C-flag One can count the numbers of *Enable* and *Disable* operations and define the flag value to be true if there are more *Enable* than *Disable* operations. This corresponds to the counter CRDT. A problem with this approach is that it is not guaranteed that the flag value is *False* after executing *Disable*. If there were multiple, possibly concurrent, *Enable* operations it is necessary to execute the same number of *Disable* operations. To circumvent this problem, it is possible to increment or decrement the counter exactly by the number that is necessary to change the value locally.

We do not formalize them here since they can be implemented using previously introduced datatypes.

Sets

In principle, a set can be seen as a collection of flags, one flag for each element. Accordingly, we have the same possibilities for resolving conflicting updates (i.e. concurrent *Add* and *Remove* of the same element).

Many of the variants discussed above can actually be found in the literature and in real world implementations. In Figure 4.7 we give an overview of the different semantics and their appearance. An overview paper by Shapiro et al [Sha+11a] describes 5 different semantics for sets. Gotsman and Yang [GY15b] give 3 different semantics for sets. Interestingly, they chose the weaker semantics for the add-wins set and the stronger for the remove-wins set. In an CRDT overview paper, Preguiça [Pre18] specifies the same variants. Baquero et al. [Baq+17] also specify an enable-wins and an remove-wins set, but choose the weaker semantics in both cases. Interestingly, it appears to be the case that papers focussing on implementations tend to use the weak semantics for the delete-wins set and papers focussing on specification tend to use the stronger variant. This might be due to the fact the weaker form allows a more efficient implementation, while the stronger form is easier to specify. For the strong add-wins semantics, we found neither an implementation nor a specification in the literature. In this case, the weaker version has an easier specification as well as a more efficient and straight-forward implementation.

In Figure 4.8 we formalize the specification for sets based on flag-semantics. To this end, we define a partial function *set-to-flag* that takes an element v and converts a set operation on element v to a corresponding flag operation. The operation *Add(v)* is converted to *Enable* and *Remove(v)* to *Disable*. For all other operations, the function is undefined.

The generic *set-spec* then takes a flag operation. The result of a *Contains(v)* query is computed by transforming the context using *restrict-ctxt-op* with the

Flag-semantics	Appearance in literature and implementations
ew	[Sha+11a; GY15b; Baq+17; Pre18]
sew	
dw	[Baq+17]
sdw	[GY15b; Pre18]
G	[Sha+11a]
2P	[Sha+11a]
lww	[GY15b; Pre18]
C	[Sha+11a]
C'	[Sha+11a]

Figure 4.7.: *Different set semantics in the literature and real implementations.*

```
datatype 'v setOp =
    Add 'v
  | Remove 'v
  | Contains 'v

definition set-to-flag where
set-to-flag v op ≡ case op of
   Add x ⇒ if x = v then Some Enable else None
 | Remove x ⇒ if x = v then Some Disable else None
 | - ⇒ None


definition set-spec :: (flagOp, 'r::{default,from-bool}) crdtSpec ⇒ ('v setOp, 'r) crdt-
Spec where
set-spec F op ctxt res ≡
  case op of
    Add - => res = default
  | Remove - ⇒ res = default
  | Contains v ⇒ F ReadFlag (restrict-ctxt-op (set-to-flag v) ctxt) res


definition set-aw-spec ≡ set-spec flag-ew-spec

definition set-rw-spec ≡ set-spec flag-dw-spec
```

Figure 4.8.: *Specification of Set CRDTs*

set-to-flag function and then applying the flag specification for *ReadFlag* on the resulting context. The function *restrict-ctxt-op* (not shown here) takes a partial function on operations and transforms the context with this function. Operations for which the function is not defined are removed from the context and its happens-before relation.

With the generic *set-spec*, we can derive the different set semantics from the flag semantics as shown in Figure 4.8 for the *aw* and *rw* variants.

Maps

In our specification framework, maps are a basic mechanism for composing CRDTs. The keys of the map are of some type $'k$ and values in the map are again CRDTs with operations of type $'v$. A basic map only supports the execution of operations on the nested value for a given key. The respective operation is *NestedOp* $'k$ $'v$. There is no explicit operation for adding entries to the map. All entries implicitly exist with their initial value from the beginning. We call this form of the map the grow-only map (*gmap*). The semantics are straight-forward: We can just create a nested operation context for each key and apply the specification of the nested CRDT on this context to get the result of an operation.

Matters become more complicated, if we consider two additional operations: An update operation *DeleteKey* $'k$ to delete an entry from the map and a query *KeyExists* $'k$ to check if an entry exists.

The existence of an entry can be seen as another *Flag* CRDT. Performing an update operation on a key enables the flag and deletion is like *Disable*. Thus, the same semantics discussed above for flags can be used again for maps. Some implementations have a more complex behavior, where the existence of an entry depends on the current value of the entry. In these implementations, an entry with an empty set as value can be considered as non-existent. For example, the implementation of Maps in Antidote[2] removes an entry, if the internal state of a value is equal to the initial state. For simplicity, we will not consider these special cases of implementations here.

However, a map semantics also needs to describe how a *DeleteKey* call affects the nested contexts used to evaluate query operations. We discuss three basic strategies here:

1. The nested context is not affected by *DeleteKey*.
2. A call to *DeleteKey* can add a special *Reset* call to the nested context.
3. Calling *DeleteKey* filters a certain subset from the nested contexts.

In alternative 1, deleting an entry from a map would just mark it as deleted but still keep the underlying value for the case that it is updated again. This can lead to a situation where a map appears to be empty, but incrementing

[2] Antidote's CRDT library is available at `https://github.com/AntidoteDB/antidote_crdt`

a counter in the map by 1 results in an entry with value 100. This does not respect the local sequential semantics of a map, which makes this choice rather unintuitive.

In alternative 2, the effect of *DeleteKey* depends on the semantics of the *Reset* operation of the nested datatype. However, there is no unified semantics for the *Reset* operation. In general, the expected behavior would be that the *Reset* operation of the nested data type filters a certain subset from the operation context, which would give us a semantics like in alternative 3 for a given composition of CRDTs. To keep the semantics unified and mostly independent of the nested datatype, we therefore focus on alternative 3 here.

To specify the shape of filtered contexts in alternative 3, we again have similar choices as we already discussed for flag CRDTs. The choices are slightly more complicated though – instead of a single Boolean result, we have to specify the set of updates that is affected by delete operations. There are two obvious choices for which updates are eliminated by deleting an entry:

uw All previous updates.

sdw All previous and concurrent updates.

The first choice corresponds to the enable-wins semantics of flags and the second to the strong disable-wins strategy on flags. The other two strategies are less obvious:

suw All previous updates, but ignore all deletes that have concurrent updates.

dw All previous updates and for the deletes, that are not followed by updates, also the concurrent updates.

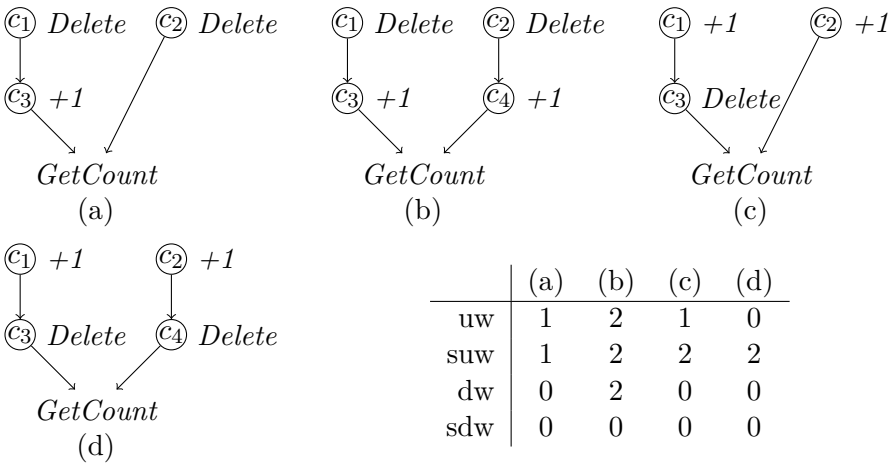
These variants correspond to the strong enable-wins and the disable-wins strategies of flags. Before we formalize these four strategies, we consider their behaviors on 4 minimal examples where we nest a counter CRDT in a map. The examples and results are shown in Figure 4.9. For brevity, we omit the keys in the example and write *Delete* for *DeleteKey(k)*, *+1* for the nested update *NestedOp(k, Increment)*, and *GetCount* for *NestedOp(k, GetCount)*.

In example (a) we have two concurrent deletions where one delete is followed by an increment. Here, both delete-wins strategies would ignore the increment operation as it is concurrent to a delete that was not followed by an update. Thus, the overall count is 0. Both update-wins strategies yield 1 since the increment is not followed by a deletion.

Example (b) shows an example where the two delete-wins strategies differ. The sdw strategy ignores both increments, since both have a concurrent deletion. The dw strategy sees that both deletions have been overridden by increments, so both increments count.

Examples (c) and (d) show the similar cases but with deletions and increments swapped. This shows the difference between the uw and suw strategies.

We now present the formalization of the map semantics. Figure 4.10 shows the generic definitions which are common for all strategies. First, we define



	(a)	(b)	(c)	(d)
uw	1	2	1	0
suw	1	2	2	2
dw	0	2	0	0
sdw	0	0	0	0

Figure 4.9.: *Different semantics for Map CRDTs, illustrated by nesting a counter in a map.*

how to compute a sub-context for a given key, where a certain subset of deleted database calls is removed from the context. The function *sub-context* takes a partial function *C-in* for transforming operations and a set of database calls Cs to which to restrict the context. The function then first restricts the context to the calls in Cs by using the function *ctxt-restrict-calls*. Then, the function *C-in* is used to transform the operations and remove all operations where *C-in* is undefined.

The concrete *C-in* function we use for maps is *nested-op-on-key(k)*. It is parameterized with a key k and transforms operations of the form *NestedOp(k, op)* to *op* and removes all other operations.

Finally, the generic *map-spec* takes two initial parameters: 1. A function that computes the set of deleted calls from a context. 2. The specification for the nested datatype. For a *KeyExists(k)* query, we check whether there is an update operation on the given key k that has not been deleted. The predicate *is-update* is part of the type class *crdt-op*.

For a nested operation on a key (*NestedOp k op*), we compute the sub-context by restricting the context to the complement of the *deleted-calls* and the *nested-op-on-key(k)* function to extract the inner operation from all operations on the given key k. We then apply the nested specification on this context.

In Figure 4.11 we then instantiate the generic *map-spec* with the four different variants we discussed earlier. This boils down to defining the set of deleted calls for each variant. In the definitions we use the abbreviation *is-concurrent ctxt c c'*, which is short for:

$$c \neq d \wedge \{(c, c'), (c', c)\} \cap happensBefore \quad ctxt = \varnothing$$

The variants *uw* and *sdw* are straight forward to formalize. For the *suw* variant, we consider a call as deleted if there exists a delete operation that happens after it and for this delete operation, there is no concurrent update operation.

```
datatype ('k,'v) mapOp =
    NestedOp 'k 'v
  | KeyExists 'k
  | DeleteKey 'k

definition restrict-ctxt-op :: ('op1 ⇀ 'op2) ⇒ ('op1, 'r) operationContext ⇒ ('op2, 'r) operationContext where
restrict-ctxt-op f ≡
  restrict-ctxt (λc.
    map-option (λop'. Call op' (call-res c)) (f (call-operation c)))

definition ctxt-restrict-calls :: callId set ⇒ ('op, 'r) operationContext ⇒ ('op, 'r) operationContext where
ctxt-restrict-calls Cs ctxt = (|
    calls = calls ctxt |` Cs,
    happensBefore = happensBefore ctxt |r Cs|)

definition sub-context :: ('c ⇒ 'a option) ⇒ callId set ⇒ ('c, 'b) operationContext ⇒ ('a, 'b) operationContext where
sub-context C-in Cs ctxt ≡
  (restrict-ctxt-op C-in (ctxt-restrict-calls Cs ctxt))

definition
nested-op-on-key k op ≡
    case op of NestedOp k' op' ⇒ if k = k' then Some op' else None
             | - ⇒ None

definition map-spec :: ((('k, 'v::crdt-op) mapOp, 'r::{default,from-bool}) operationContext ⇒ 'k ⇒ callId set) ⇒ ('v,'r) crdtSpec ⇒ (('k, 'v) mapOp, 'r) crdtSpec where
map-spec deleted-calls nestedSpec oper ctxt res ≡
  case oper of
    DeleteKey k ⇒ res = default
  | KeyExists k ⇒ res = from-bool (∃ c op r. calls ctxt c ≐ Call (NestedOp k op) r
                                   ∧ is-update op ∧  c ∉ deleted-calls ctxt k)
  | NestedOp k op ⇒
     nestedSpec op (sub-context (nested-op-on-key k) (- deleted-calls ctxt k) ctxt) res
```

Figure 4.10.: *Infrastructure for specification of Map CRDTs*

definition
deleted-calls-uw ctxt k ≡ {*c*∈*dom* (*calls ctxt*).
∃ *d*. *Op ctxt d* ≜ *DeleteKey k* ∧ (*c*,*d*)∈*happensBefore ctxt*}

definition
deleted-calls-suw ctxt k ≡ {*c*∈*dom* (*calls ctxt*).
∃ *d*. *Op ctxt d* ≜ *DeleteKey k* ∧ (*c*,*d*)∈*happensBefore ctxt*
∧ (∄ *u u-op*. *Op ctxt u* ≜ *NestedOp k u-op* ∧ *is-update u-op* ∧ *is-concurrent ctxt u d*)}

definition
deleted-calls-dw ctxt k ≡ {*c*∈*dom* (*calls ctxt*).
∃ *d*. *Op ctxt d* ≜ *DeleteKey k* ∧
((*c*,*d*)∈*happensBefore ctxt*
∨ *is-concurrent ctxt c d*
∧ (∄ *u u-op*. *Op ctxt u* ≜ *NestedOp k u-op* ∧ *is-update u-op*
∧ (*d*,*u*)∈*happensBefore ctxt*))}

definition
deleted-calls-sdw ctxt k ≡ {*c*∈*dom* (*calls ctxt*).
∃ *d*. *Op ctxt d* ≜ *DeleteKey k* ∧ (*d*,*c*)∉*happensBefore ctxt*}

definition *map-uw-spec* :: (*'v*::*crdt-op*,*'r*::{*default*,*from-bool*}) *crdtSpec* ⇒ ((*'k*, *'v*) *mapOp*, *'r*) *crdtSpec* **where**
map-uw-spec ≡ *map-spec deleted-calls-uw*

definition *map-suw-spec* :: (*'v*::*crdt-op*,*'r*::{*default*,*from-bool*}) *crdtSpec* ⇒ ((*'k*, *'v*) *mapOp*, *'r*) *crdtSpec* **where**
map-suw-spec ≡ *map-spec deleted-calls-suw*

definition *map-dw-spec* :: (*'v*::*crdt-op*,*'r*::{*default*,*from-bool*}) *crdtSpec* ⇒ ((*'k*, *'v*) *mapOp*, *'r*) *crdtSpec* **where**
map-dw-spec ≡ *map-spec deleted-calls-dw*

definition *map-sdw-spec* :: (*'v*::*crdt-op*,*'r*::{*default*,*from-bool*}) *crdtSpec* ⇒ ((*'k*, *'v*) *mapOp*, *'r*) *crdtSpec* **where**
map-sdw-spec ≡ *map-spec deleted-calls-sdw*

Figure 4.11.: *Specification of concrete Map CRDTs*

In the *dw* variant we distinguish two cases how a call c can be considered deleted by a *DeleteKey* operation d: 1. The call c is before d. 2. The calls c and d are concurrent and there is no update operation after d.

Structs

Like a map, a struct also is a composition of multiple CRDT instances. The difference is that a map is a homogenous composition – all values in the map must be of the same type. A struct can have fields with different types, yet the fields are static whereas a map supports dynamic addition and removal of entries.

Unfortunately, the possibility to have different types in the same struct prevents us from giving the specification of structs in the same way as the previously introduced datatypes. Instead, we merely provide an auxiliary function named *struct-field* that can be used to specify a structs' semantics. With the help of this function, a new struct can be defined with the following steps.

1. Define a datatype representing the operations on the struct. The datatype has one case for every field of the struct and each case has one parameter, which is the operation type for the field.

   For example, if we want a struct with a field A containing a counter CRDT and a field B containing a set of integers, we would define the following datatype:

   datatype *structOp* =
   A counterOp
   | *B* ‹*int setOp*›

2. Define the specification for the struct by using the *struct-field* helper function. This function takes the datatype constructor of the field, which was defined in step 1, as the first parameter and the specification for the field as the second parameter. Multiple fields can be combined with the $(. \vee .)$ operator.

   definition *crdtSpec* :: (*structOp*, *val*) *crdtSpec* **where**
   crdtSpec ≡
   struct-field A counter-spec
   .∨. *struct-field B set-rw-spec*

The relevant building blocks for structs are defined in Figure 4.12. First we have the definition of *select-field*, which takes a unary function and inverts it, returning *None* if no inverse exists. We use this function to define *struct-field*, which takes the constructor of the field (type $'i \Rightarrow' o$) and the specification of the field. The spec returns *False* for operations that are not part of the field. For other operations, it restricts the context to all operations nested under the field and applies the nested specification.

Since *struct-field* specifications returns *False* for other fields, we can compose disjoint fields by simply taking the disjunction of multiple specifications, which is how the operator $(. \vee .)$ is defined.

```
definition select-field f x ≡
  if ∃ y. x = f y then Some (inv f x) else None

definition struct-field :: ('i ⇒ 'o) ⇒ ('i, 'r) crdtSpec ⇒ ('o, 'r) crdtSpec  where
struct-field f spec ≡ λop ctxt r.
    case select-field f op of
      Some i-op ⇒ spec i-op (restrict-ctxt-op (select-field f) ctxt) r
    | None ⇒ False

definition compose-struct :: ('o, 'r) crdtSpec ⇒ ('o, 'r) crdtSpec ⇒ ('o, 'r) crdtSpec
(infixr .∨. 30) where
A .∨. B ≡ λop ctxt r. A op ctxt r ∨ B op ctxt r
```

Figure 4.12.: *Specification of Struct CRDTs*

```
type-synonym ('op, 'opn, 'res) cOperationResultSpec =
      callId set                  — visible calls
    ⇒ (callId ⇒'op)               — call information
    ⇒ callId rel                  — happens-before
    ⇒ ('opn ⇒ 'op)                — mapping back
    ⇒ 'res
    ⇒ bool

type-synonym ('op, 'opn, 'res) ccrdtSpec =
      'opn ⇒ ('op, 'opn, 'res) cOperationResultSpec
```

Figure 4.13.: *Type definitions for first-order CRDT specifications.*

4.2. First-order CRDT Specifications

The CRDT specifications we have considered so far cannot be easily expressed with first-order formulas, which is important for automating verification with the help of automated theorem provers. The main problem lies in the composition of CRDTs in maps, where the map specification takes the specification of the nested type as a parameter and applies it on a sub-context. The way in which sub-contexts are computed makes it hard to transform the nested specification so that it again works on the outer context, which would make the specification first-order again.

We address this problem with a different format for specifications, which is less elegant for specifying CRDTs, but leads to simpler formulas in the case of nested specifications. The type definitions for these specifications are given in Figure 4.13.

Given an operation, the type *cOperationResultSpec* is the predicate that determines the possible results for a given context. The specification depends on three type parameters:

1. *'op* is the type of top-level operations.
2. *'opn* is the type of nested operations.

```
definition crdt-spec-rel :: ('opn, 'res) crdtSpec ⇒ ('op, 'opn, 'res) ccrdtSpec ⇒ bool
where
crdt-spec-rel spec cspec ≡
∀ C-in::'op ⇀ 'opn. ∀ C-out::'opn ⇒ 'op.
  is-reverse C-in C-out
  ⟶
  (∀ ctxt (outer-op::'op) (op::'opn) r Cs.
      C-in outer-op ≜ op
    ⟶ Cs ⊆ (dom (calls ctxt))
    ⟶ operationContext-wf ctxt
    ⟶
      (spec op (sub-context C-in Cs ctxt) r
    ⟷ cspec op Cs (extract-op (calls ctxt)) (happensBefore ctxt) C-out r))
```

Figure 4.14.: *Relation between first-order and higher-order CRDT specifications.*

3. *'res* is the result type.

The presence of two operation types hints at the first difference compared to previous specifications. The specifications describe the nested operations, but they work on the top-level context which has operations of type *'op*.

In the previous specifications, we used an operation context that only included the call information and the happens-before relation. We restricted both to the set of visible calls when applying the specification. In the first-order specifications, we get rid of the need to restrict the context. Instead, we include the set of visible calls as an explicit parameter and pass the full happens-before relation and the information about all call operations. We omit the results of call operations since it is not needed for most specifications.

Finally, we have one additional parameter of type *'opn ⇒ 'op* which allows us to map nested operations back to the top-level context.

Relating Specifications. We now show how to relate the first-order specification style to our previous specifications. To this end, we define the predicate *crdt-spec-rel* which is given in Figure 4.14 and relates two CRDT specifications given in the two different formats.

For two specifications to be considered equivalent, they must behave equivalently in all nested contexts. This means that the equivalence must hold for any mapping functions *C-in* and *C-out*, where *C-in* maps operations from the outer context to the nested operations and *C-out* is the reverse function. We then quantify over all well-formed operation contexts. Here, an operation context is considered well-formed if it satisfies the following conditions:

1. The happens-before relation must be transitive and irreflexive.

2. The happens-before relation may only relate calls that exist in the context, i.e. the field of the relation must be a subset of the domain of the context's call map.

```
lemma set-rw-spec-Contains:
  assumes spec: set-rw-spec (Contains x) ctxt res
    and wf: operationContext-wf ctxt
  shows res = from-bool ((∃ a. Op ctxt a ≜ Add x)
                         ∧ (∀ r. Op ctxt r ≜ Remove x
                              ⟶ (∃ a. Op ctxt a ≜ Add x
                                   ∧ (r,a)∈happensBefore ctxt)))

definition set-rw-spec' :: ('op, 'v setOp, ('r::{default,from-bool})) ccrdtSpec where
set-rw-spec' oper vis op hb C  res ≡
  case oper of
    Add - => res = default
  | Remove - ⇒ res = default
  | Contains v ⇒ res = from-bool
       (∃ a∈vis. op a = C (Add v)
         ∧ (∀ r∈vis. op r = C (Remove v)
              ⟶ (∃ a'∈vis. op a' = C (Add v) ∧ (r,a')∈hb)))
```

Figure 4.15.: *Comparison of different specifications of the Remove-wins Set CRDT. The original specification is given above, the first-order specification in the definition below.*

3. There must only be finitely many calls in the context.

Finally, we take some subset of calls *Cs* from the set of all calls in the context, such that all calls in *Cs* map into the nested context via *C-in*. We then demand that the two specifications yield the same result, when we use *Cs* to calculate a sub-context for the higher-order specification or when we use *Cs* as a parameter to the first-order specification with the original call and happens-before data.

Note that the higher-order specification is only passed the information related to the calls in *Cs* but the higher-order specification is passed the complete information. Thus, a first-order specification can only be in relation with a higher-order specification if it internally discards the extra information it is given and only accesses the operations and happens-before information for calls in *Cs*.

Example: Remove-wins Set

To illustrate the style of first-order specifications, we now consider the specification of the remove-wins set CRDT given in Figure 4.15.

As a first step, we simplify the original definition (see Figure 4.8 on page 34), which was based on flag CRDTs. In Lemma *set-rw-spec-Contains* we remove this dependency on the flag semantics and express the specification of the *Contains* query directly. Recall that the flag semantic was based on the latest operations, i.e. operations that have not been overridden by other updates. We defined that the result is true if there is an *Enable*-operation and no *Disable* operation in the set of latest operations. We can translate

this to the set query *Contains(x)* by stating that there is at least one *Add(x)* operation in the context and all *Remove(x)* operations have been overridden by a later call to *Add(x)*. To prove this equality, we need the well-formedness of the context. In particular, we need the finiteness of the context since for infinite contexts the set of latest operations might be empty, in which case the two specifications would differ.

In the second step, we transform the specification to the first-order format. The basic structure of the formula is the same. There are just a few differences:

1. When quantifying over database calls, we now quantify over the set of visible calls (*vis*). In the original specification, this aspect is captured by the fact that *Op* is a partial function that is only defined for visible calls. In first-order specifications, *op* is a total function, so we cannot use the $\triangleq$ comparison and need to guard every use of *op* with a check that the call is visible.

2. Since *Op* in the original specifications contains the set operations at the outermost level, we can directly use operations like *Add* in the specification. In the first-order specifications, we instead have to transform all set-operations to the outermost level by calling the conversion function *C*.

Example: Delete-wins Map

We now consider the sdw-map specification in Figure 4.16 as an example of an CRDT with nesting. This specification corresponds to the original specification in Figure 4.11 on page 39.

First, we define the function *restrict-calls*, which restricts the set of visible calls to only those calls that are nested operations on a given key. This is simpler than in the original definition, since we only need to restrict the visible calls instead of the complete context.

Then we define the generic map specification *map-spec'*. Like in the original definition this depends on a parameter to calculate the deleted calls and a parameter for the nested specification. For a *KeyExists* query we check that there exists an update that has not been deleted. To define *NestedOp*, we simply call the nested specification, where we restrict the set of visible calls to the operations on the given key minus the set of deleted calls. For the conversion function, we use the C from the map itself and add another level of *NestedOp* to it.

Finally, we can use the above definition to define a concrete strategy. The definition of *deleted-calls-sdw'* is similar to the original definition, but uses the parameter C to shift operations to the outermost context.

Example: Generated Verification Conditions

Let us now again consider our running example of the chat application to see how the generated verification conditions differ between the two styles of CRDT specifications. The CRDT in the chat application consists of a

definition
restrict-calls vis op C k ≡
{*c*∈*vis*. ∃ *u*. *op c* = *C* (*NestedOp k u*) }

definition *map-spec′* ::
(*callId set* ⇒ (*callId* ⇒′*op*) ⇒ *callId rel* ⇒ ((′*k*, ′*opn*) *mapOp* ⇒ ′*op*) ⇒ ′*k* ⇒ *callId set*)
⇒ (′*op*, ′*opn*::*crdt-op*, ′*r*) *ccrdtSpec*
⇒ (′*op*, (′*k*, ′*opn*) *mapOp*, (′*r*::{*default*,*from-bool*})) *ccrdtSpec* **where**
map-spec′ deleted-calls nestedSpec oper vis op hb C res ≡
case oper of
DeleteKey k ⇒ *res* = *default*
| *KeyExists k* ⇒ *res* = *from-bool* (∃ *c*∈*vis*. ∃ *upd-op*. *op c* = *C* (*NestedOp k upd-op*) ∧ *is-update upd-op* ∧ *c* ∉ *deleted-calls vis op hb C k*)
| *NestedOp k nested-op* ⇒
nestedSpec nested-op (*vis* − *deleted-calls vis op hb C k*) *op hb* (λ*x*. *C* (*NestedOp k x*)) *res*

definition *deleted-calls-sdw′* :: *callId set* ⇒ (*callId* ⇒′*op*) ⇒ *callId rel* ⇒ ((′*k*, ′*opn*) *mapOp* ⇒ ′*op*) ⇒ ′*k* ⇒ *callId set* **where**
deleted-calls-sdw′ vis op hb C k ≡ {*c*∈*vis*. ∃ *d*∈*vis*. *op d* = *C* (*DeleteKey k*) ∧ (*d*,*c*)∉*hb*}

definition
map-sdw-spec′ ≡ *map-spec′ deleted-calls-sdw′*

Figure 4.16.: *First-order specification of the sdw-map.*

obtain a
  where $a \in vis$
    and $\forall\, d{\in}vis.\ op\ d = Message\ (DeleteKey\ msg) \longrightarrow (d,\ a) \in hb$
    and $op\ a = Message\ (NestedOp\ msg\ (Author\ (Assign\ (String\ author))))$
    and $\forall\, c'.\ c' \in vis \longrightarrow$
        $(\exists\, d{\in}vis.\ op\ d = Message\ (DeleteKey\ msg) \wedge (d,\ c') \notin hb)$
      $\vee\ (\forall\, v'.\ op\ c' \neq Message\ (NestedOp\ msg\ (Author\ (Assign\ v'))))$
      $\vee\ (a,\ c') \notin hb$

Figure 4.17.: *Resulting formula for first-order specifications.*

sub-ctxt ≡
  (*restrict-ctxt-op* (*select-field Author*)
      (*sub-context* (*nested-op-on-key msg*)
        (− *deleted-calls-sdw* (*restrict-ctxt-op* (*select-field Message*)
                                (⦇*calls* = *cs*, *happensBefore* = *hb*⦈)) *msg*)
        (*restrict-ctxt-op* (*select-field Message*)
                                (⦇*calls* = *cs*, *happensBefore* = *hb*⦈))))

obtain c **where** *Op sub-ctxt* $c \triangleq$ *Assign* (*String author*)
and $\forall\, c'.\ (\forall\, v'.\ Op\ sub\text{-}ctxt\ c' \neq Some\ (Assign\ v'))$
  $\vee\ (c,\ c') \notin happensBefore\ sub\text{-}ctxt$

Figure 4.18.: *Resulting formula for higher-order specifications.*

struct, where one field stores the messages in a map from message identifier to another struct, which contains registers for the author and the content of the message. In this context, we consider the case where reading the author field of a message *msg* returns the string *author*:

crdtSpec′ (*Message* (*NestedOp msg* (*Author Read*))) *vis op hb id* (*String author*)

This query comes up in the verification of the *getMessage* procedure. If we take this query and unfold all the specifications, we get to the facts shown in Figure 4.17. There are at the level of the database history. We are able to obtain a database call a which is visible and has the operation that assigned the string *author* to the register. Moreover, we know that all visible calls that would delete the entry for our the message *msg* from the map happened before a. Finally, we can derive that the assignment a has not been overridden, so any other visible call c' is either deleted, is no assignment to the register, or does not come after a.

This example shows that by simply unfolding the relevant definitions, we can get a succinct representation in terms of the database history, which enables us to easily reason about the results. On the other hand, the original higher-order specifications do not produce these nice results. Figure 4.18 shows the resulting facts after unfolding the basic definitions but before unfolding the sub-context generated for the query.

We can already see that the formula for the sub-context is quite complicated. As soon as we try to unfold the definitions related to this sub-context

the increases substantially, so we cannot show it here on one page. All definitions transforming the context work with functions, but since we have no information about the inverse of these functions, their composition cannot be automatically simplified to a similar representation as we have seen for the first-order specifications.

Chapter 5

A Formalized Proof Technique

In this chapter we present a formalized approach for verifying the correctness of highly available applications. All definitions and proofs in this chapter have been formalized and checked in Isabelle/HOL [NPW02]. The Isabelle theories are available on Github[1].

To precisely define correctness of applications, we introduce an operational, small-step, interleaving semantics in Section 5.1. In principle, we could directly verify an application in Isabelle using these definitions. However, without a more systematic approach, this is very tedious. The goal of the Repliss proof technique is to simplify the verification problem.

To handle the fine-grained concurrency we reduce the verification problem to the verification against a simplified semantics, which we call the single-invocation semantics. This semantics only considers a single procedure invocation and uses invariants to reason about the effects of other invocations. With this step, we eliminate the need to reason about fine-grained concurrent interactions. We describe this single-invocation semantics and its relation to the interleaving semantics in Section 5.2. In Section 5.3, we present our soundness proof which shows that we can use the simpler semantics for verification. Finally, in Section 5.4 we discuss completeness of this reduction.

5.1. Interleaving Semantics

In this section, we formalize the system semantics in the form of an operational small-step semantics. Concurrency is handled by interleaving of actions of different procedure invocations. In the following we therefore call this semantics the *interleaving semantics*. Procedures define the external interface (API) of a program. Clients invoke the procedures of a program which creates a new concurrent process. As there is no other mechanism to introduce concurrency, there is a one-to-one correspondence between concurrent processes and procedure invocations. Each procedure invocation is executed as a sequential process, interleaved with the other concurrent invocations.

[1] `https://github.com/peterzeller/repliss-isabelle`

Figure 5.1 shows the definitions regarding system state that we use in our formalization. The semantics is parameterized by four type parameters:

'op The type of operations that can be performed on the database.

'proc The procedures provided by the application. The type includes the procedure arguments.

'any The type of values used by programs, which is also the type of values returned by database calls and procedure invocations.

'ls The local state of a procedure invocation. At this step we do not assume a fixed programming language in which procedures are implemented. Instead, we model a procedure implementation using an abstract state transition system with states of type *'ls*. Later, in Section 6.1 we instantiate this abstract state machine with programs written in a concrete programming language.

The system state is organized as a hierarchy of records, which allows us to precisely restrict a definition to certain parts of the state.

The first definition is the *operationContext*, which represents the database state and thus is also the only part of the state that is used in database query specifications later on. An operation context comprises a map containing the operation and result of each database calls and a happens-before relation on the calls.

The *operationContext* is extended to an *invContext*, which is the part of the state that can be addressed by invariants. Here, we also have information about database transactions, the procedure invocation history, and about unique identifiers.

Finally, *state* defines the remaining fields. Here, we include the status of transactions, the generated unique identifiers, and the local state for each procedure invocation. We will explain these fields in detail below, together with the rules for the semantics.

The rules of our semantics are shown in Figure 5.2 and 5.3. We write $S \xrightarrow{i,a} S'$ to denote that the system makes a step from state S to S' by executing action a in procedure invocation i. In every step, a different invocation can progress, resulting in a fine-grained interleaving semantics. $S \xrightarrow{tr}{}^{*} S'$ denotes the reflexive, transitive closure with the trace tr. The trace is the sequence of (*invocId*, *action*) pairs of the individual steps.

Each rule describes the complete effect of a single action which includes some orthogonal aspects of our semantics. In the following we therefore describe the different aspects and how they manifest in the rules.

Procedure invocations. In our semantics, a procedure invocation is triggered by an application request from some client. Clients may invoke procedures concurrently, but each single invocation executes sequentially.

Programs (record *prog*) are modeled with a function (field *procedure*) that takes the procedure name and the arguments (*'proc*) of the invocation and

```
record ('op, 'any) operationContext =
  calls :: callId ⇀ ('op, 'any) call
  happensBefore :: callId rel

record ('proc, 'op, 'any) invContext = ('op, 'any) operationContext +
  callOrigin :: callId ⇀ txId
  txOrigin :: txId ⇀ invocId
  knownIds :: uniqueId set
  invocOp :: invocId ⇀ 'proc
  invocRes :: invocId ⇀ 'any

record ('proc, 'ls, 'op, 'any) state = ('proc, 'op, 'any) invContext +
  prog :: ('proc, 'ls, 'op, 'any) prog
  txStatus :: txId ⇀ txStatus
  generatedIds :: uniqueId ⇀ invocId
  localState :: invocId ⇀ 'ls
  currentProc :: invocId ⇀ ('ls, 'op, 'any) procedureImpl
  visibleCalls :: invocId ⇀ callId set
  currentTx :: invocId ⇀ txId

record ('proc, 'ls, 'op, 'any) prog =
  querySpec :: 'op ⇒ ('op, 'any) operationContext ⇒ 'any ⇒ bool
  procedure :: 'proc ⇒ ('ls × ('ls, 'op, 'any) procedureImpl)
  invariant :: ('proc, 'op, 'any) invContext ⇒ bool

datatype ('op, 'any) call = Call (call-operation: 'op) (call-res:'any)

type-synonym ('ls, 'op, 'any) procedureImpl =
  'ls ⇒ ('ls, 'op, 'any) localAction

datatype ('ls, 'op, 'any) localAction =
    LocalStep bool 'ls
  | BeginAtomic 'ls
  | EndAtomic 'ls
  | NewId 'any ⇀ 'ls
  | DbOperation 'op 'any ⇒ 'ls
  | Return 'any

datatype ('proc, 'op, 'any) action =
    ALocal bool
  | ANewId 'any
  | ABeginAtomic txId (callId set)
  | AEndAtomic
  | ADbOp callId 'op 'any
  | AInvoc 'proc
  | AReturn 'any
  | ACrash
  | AInvcheck bool
```

Figure 5.1.: *Type definitions for the formalizing the system semantics.*

invocation:

$$\begin{array}{l} localState\ S\ i = None \wedge \\ procedure\ (prog\ S)\ proc = (initialState,\ impl) \wedge \\ uniqueIds\ proc \subseteq knownIds\ S \wedge invocOp\ S\ i = None \Longrightarrow \\ S \xrightarrow{(i,\ AInvoc\ proc)} S(\!|localState := localState\ S(i \mapsto initialState), \\ \qquad\qquad\qquad\qquad currentProc := currentProc\ S(i \mapsto impl), \\ \qquad\qquad\qquad\qquad visibleCalls := visibleCalls\ S(i \mapsto \varnothing), \\ \qquad\qquad\qquad\qquad invocOp := invocOp\ S(i \mapsto proc)|\!) \end{array}$$

return:

$$\begin{array}{l} localState\ S\ i \triangleq ls \wedge \\ currentProc\ S\ i \triangleq f \wedge f\ ls = Return\ res \wedge currentTx\ S\ i = None \Longrightarrow \\ S \xrightarrow{(i,\ AReturn\ res)} S(\!|localState := (localState\ S)(i := None), \\ \qquad\qquad\qquad\qquad currentProc := (currentProc\ S)(i := None), \\ \qquad\qquad\qquad\qquad visibleCalls := (visibleCalls\ S)(i := None), \\ \qquad\qquad\qquad\qquad invocRes := invocRes\ S(i \mapsto res), \\ \qquad\qquad\qquad\qquad knownIds := knownIds\ S \cup uniqueIds\ res|\!) \end{array}$$

local:

$$\begin{array}{l} localState\ S\ i \triangleq ls \wedge currentProc\ S\ i \triangleq f \wedge f\ ls = LocalStep\ ok\ ls' \Longrightarrow \\ S \xrightarrow{(i,\ ALocal\ ok)} S(\!|localState := localState\ S(i \mapsto ls')|\!) \end{array}$$

newId:

$$\begin{array}{l} localState\ S\ i \triangleq ls \wedge \\ currentProc\ S\ i \triangleq f \wedge \\ f\ ls = NewId\ ls' \wedge \\ generatedIds\ S\ uid = None \wedge \\ uniqueIds\ uidv = \{uid\} \wedge ls'\ uidv \triangleq ls'' \wedge uid = to\text{-}nat\ uidv \Longrightarrow \\ S \xrightarrow{(i,\ ANewId\ uidv)} S(\!|localState := localState\ S(i \mapsto ls''), \\ \qquad\qquad\qquad\qquad generatedIds := generatedIds\ S(uid \mapsto i)|\!) \end{array}$$

Figure 5.2.: *Interleaving semantics, Part 1.*

beginAtomic:

$$\begin{array}{l}
localState\ S\ i \triangleq ls \wedge \\
currentProc\ S\ i \triangleq f \wedge \\
f\ ls = BeginAtomic\ ls' \wedge \\
currentTx\ S\ i = None \wedge \\
txStatus\ S\ t = None \wedge \\
visibleCalls\ S\ i \triangleq vis \wedge chooseSnapshot\ snapshot\ vis\ S \Longrightarrow \\
S \xrightarrow{(i,\ ABeginAtomic\ t\ snapshot)} S(\!|localState := localState\ S(i \mapsto ls'), \\
\qquad\qquad\qquad\qquad currentTx := currentTx\ S(i \mapsto t), \\
\qquad\qquad\qquad\qquad txStatus := txStatus\ S(t \mapsto \\
Uncommitted), \\
\qquad\qquad\qquad\qquad txOrigin := txOrigin\ S(t \mapsto i), \\
\qquad\qquad\qquad\qquad visibleCalls := visibleCalls\ S(i \mapsto \\
snapshot)|\!)
\end{array}$$

endAtomic:

$$\begin{array}{l}
localState\ S\ i \triangleq ls \wedge \\
currentProc\ S\ i \triangleq f \wedge f\ ls = EndAtomic\ ls' \wedge currentTx\ S\ i \triangleq t \Longrightarrow \\
S \xrightarrow{(i,\ AEndAtomic)} S(\!|localState := localState\ S(i \mapsto ls'), \\
\qquad\qquad\qquad\qquad currentTx := (currentTx\ S)(i := None), \\
\qquad\qquad\qquad\qquad txStatus := txStatus\ S(t \mapsto Committed)|\!)
\end{array}$$

dbop:

$$\begin{array}{l}
localState\ S\ i \triangleq ls \wedge \\
currentProc\ S\ i \triangleq f \wedge \\
f\ ls = DbOperation\ Op\ ls' \wedge \\
currentTx\ S\ i \triangleq t \wedge \\
calls\ S\ c = None \wedge \\
querySpec\ (prog\ S)\ Op\ (getContext\ S\ i)\ res \wedge visibleCalls\ S\ i \triangleq vis \Longrightarrow \\
S \xrightarrow{(i,\ ADbOp\ c\ Op\ res)} S(\!|localState := localState\ S(i \mapsto ls'\ res), \\
\qquad\qquad\qquad\qquad calls := calls\ S(c \mapsto Call\ Op\ res), \\
\qquad\qquad\qquad\qquad callOrigin := callOrigin\ S(c \mapsto t), \\
\qquad\qquad\qquad\qquad visibleCalls := visibleCalls\ S(i \mapsto vis \cup \{c\}), \\
\qquad\qquad\qquad\qquad happensBefore := happensBefore\ S \cup vis \times \{c\}|\!)
\end{array}$$

crash:

$$\begin{array}{l}
localState\ S\ i \triangleq ls \Longrightarrow \\
S \xrightarrow{(i,\ ACrash)} S(\!|localState := (localState\ S)(i := None), \\
\qquad\qquad\qquad\qquad currentTx := (currentTx\ S)(i := None), \\
\qquad\qquad\qquad\qquad currentProc := (currentProc\ S)(i := None), \\
\qquad\qquad\qquad\qquad visibleCalls := (visibleCalls\ S)(i := None)|\!)
\end{array}$$

invCheck:

$$invariant\text{-}all\ S = res \Longrightarrow S \xrightarrow{(i,\ AInvcheck\ res)} S$$

Figure 5.3.: *Interleaving semantics, Part 2.*

returns the initial local state of the procedure and its implementation. The implementation is represented by an abstract state machine with states of type *'ls*. The implementation (*procedureImpl*) is given by a function which calculates the next action based on the current invocation state. Possible actions are represented by the type *localAction* and comprise local evaluations (*local*), generating unique identifiers (*newId*), database related actions (*beginAtomic*, *endAtomic*, *dbOp*) and returning from an invocation (*return*). In the system state, we use the fields *currentProc* to store the implementation and the field *localState* to store the local invocation state.

The rule *invocation* describes the start of a procedure invocation. The precondition of the rule enforces that the procedure is defined for the given arguments. The remaining aspects of this rule are related to tracking the history and handling of unique identifiers (see below). The *return* rule is similar.

For local actions in a procedure invocation we have the rule (*local*), which represents local computations and state changes in a programming language, for example changing variable values. Local actions may fail, which we denote by the *ok* flag in the rule being *False*. This allows us to model local assertions as well as runtime errors, like dereferencing an invalid reference.

Database operations. Instead of modeling the database state explicitly and thus assuming a concrete implementation of the database, we represent the current state using event graphs of the database calls, as described in Chapter 4. A program has a CRDT specification of the format introduced in that chapter.

In the formal model, each database call is identified by a *callId*; the partial function *call* in the system state stores information about each call. The *callInfo* consists of the arguments to the operation and its return value. The *happensBefore* relation records the partial order between calls and *callOrigin* stores the transaction each call originated from. Transactions are identified by a *txId*, and its *txStatus* can be *uncommitted* or *committed*. The procedure invocation that started a transaction is stored in *txOrigin*. Together, this information represents the history of database calls. Additionally, the field *visibleCalls* keeps the set of database calls that are visible at a procedure invocation and *currentTransaction* the currently running transaction (if any) for an invocation.

The rule *beginAtomic* describes what happens on the database when starting a new transaction. The rule picks a fresh transaction identifier t. In the new state the status for t is set to *uncommitted*, the current transaction is changed to t for invocation i, and we record the origin of the transaction as i.

Recall that we here consider databases in which transactions work on causally consistent snapshots, modeled as a set of visible updates on the database. At the start of transaction, a snapshot is chosen. The corresponding *chooseSnapshot* predicate is defined in Figure 5.4. To obtain the snapshot, we choose an arbitrary set of committed transactions *newTxns* which are to become visible. From this we get the set of newly visible calls *newCalls*, by taking the calls in

$$
\begin{array}{l}
\mathit{chooseSnapshot}\ \mathit{snapshot}\ \mathit{vis}\ S = \\
(\exists\, \mathit{newTxns}\ \mathit{newCalls}. \\
\quad (\forall\, \mathit{txn} \in \mathit{newTxns}.\ \mathit{txStatus}\ S\ \mathit{txn} \triangleq \mathit{Committed}) \wedge \\
\quad \mathit{newCalls} = \mathit{callsInTransaction}\ S\ \mathit{newTxns} \downarrow \mathit{happensBefore}\ S \wedge \\
\quad \mathit{snapshot} = \mathit{vis} \cup \mathit{newCalls})
\end{array}
$$

$$
(x \in S \downarrow R) = (x \in S \vee (\exists\, y \in S.\ (x,\ y) \in R))
$$

$$
\mathit{callsInTransaction}\ S\ \mathit{newTxns} = \{c \mid \exists\, \mathit{txn} \in \mathit{newTxns}.\ \mathit{callOrigin}\ S\ c \triangleq \mathit{txn}\}
$$

Figure 5.4.: *Choosing transaction snapshts.*

the chosen transactions and calculating the downwards closure (denoted by ↓) with respect to the happens-before relation. In our model, the happens-before relation is by construction transitive, so that the definition of the downwards-closure itself does not need to include transitively related elements. The new snapshots then is the union of the old snapshot *vis* and the newly added calls.

When ending the current transaction (rule *commit*), its *txStatus* is set to *committed*. This allows it to be included in new snapshots, which eventually makes the database calls in the transaction visible to others. As the atomic rule can only pick committed transactions and no new calls can be added to it after committing, transactions are atomic.

When executing a database operation (rule *DB-operation*), we extract the *operationContext* from the current state using the *getContext* function. As described in Chapter 4, the operation context consists of the currently visible calls and the happens-before relation restricted to the visible calls. Formally, if $\mathit{visibleCalls}\ S\ i \triangleq \mathit{vis}$ then *getContext* is defined as:

$$
\begin{array}{l}
\mathit{getContext}\ S\ i = \\
(\!|\mathit{calls} = \mathit{calls}\ S{\restriction_{\mathit{vis}}},\ \mathit{happensBefore} = \mathit{happensBefore}\ S\ |r\ \mathit{vis}|\!)
\end{array}
$$

We then nondeterministically pick a result *res*, which satisfies the query specification (*querySpec*) of the program in the operation context. The database call is then recorded in the state by adding the operation with its arguments and result to the existing calls. We also record the current transaction as the originating transaction for the new call. The happens-before relation is also updated by making the new call causally depend on all currently visible calls. The new call is then added to the set of visible calls, such that following operations depend on it.

History Recording. As for the database, we also store a history of invocations of API-procedures including the respective arguments (*invocationOp*) and the result for completed invocations (*invocationRes*). By using these in specifications, we can relate different procedure invocations and link procedure invocation with their corresponding changes in the database state. The rules *invocation* and *return* update this information accordingly.

Unique Identifiers. In practice, unique identifiers are often generated using UUIDs or using a replica-specific identifier together with a locally unique identifier. For example, the replicated database Cassandra includes a type named *timeuuid* and a function *now* that guarantees to generate a globally unique value[2].

Since identifiers for database entries appear in most applications, we include a builtin action, which lets applications generate globally unique identifiers (see rule *new-id*). With this extension, we avoid proving the correctness of an identifier generator for every application. Moreover, it allows us to handle generated identifiers as special values, which cannot be forged by clients.

To model unique identifiers, we require that the type *any* comes with a function $uniqueIds : any \to uniqueId\ set$ extracting the unique identifiers of a value. A *uniqueId* is simply modelled as a natural number. The *new-id* rule ensures that the generated value includes exactly one unique identifier, which is the same number as we get when converting the generated value to a natural number using *to-nat*. The *NewId* action takes a parameter *ls'*, which is a partial function that takes the newly generated identifier and returns the next local state. The *new-id* rule demands that the generated identifier must be in the domain of the *ls'* function. This allows us to include a kind of type-check in the action to generate a unique identifier of a specific form.

To describe the semantics, we keep track of all generated unique identifiers in *generatedIds*. The set *knownIds* represents the identifiers which could be known to clients, i.e. identifiers which have been returned from an invocation of the application API (see rule *return*). In the *invocation* rule, we enforce that clients can only invoke the API with known identifiers.

Partial Failures. Since we are considering a distributed application, it is necessary to handle partial failures. We are following the crash-stop failure model [CGR11] in our model. This is captured in the semantics with the rule *crash*, which models a crash of a single procedure invocation and loses all locally stored information. Afterwards, the invocation cannot continue, since there is no local state.

Invariants. We use invariants to specify the application. Invariants can refer to the database state as well as the history of procedure invocations, which makes the specification language more expressive than related work where invariants can only refer to the database state.

Formally, rule *inv* is always enabled so the invariant can be checked (and therefore must hold) at all times. However, the invariant cannot involve arbitrary aspects of the current state. The function *invariant-all* checks the invariant in an invariant context that is restricted to the set of committed calls. All information local to a particular procedure invocation (i.e. *txStatus*, *generatedIds*, *localState*, *currentProc*, *currentTransaction*, and *visibleCalls*) is

[2] Cassandra documentation: UUID and timeuuid functions `https://docs.datastax.com/en/archived/cql/3.3/cql/cql_reference/timeuuid_functions_r.html`

not included in the invariant context. These restrictions are essential for simplifying the verification efforts, which we discuss in the next section. Formally, the *invariant-all* predicate is defined as:

$$
\begin{aligned}
&\mathit{invariant\text{-}all}\ S = \\
&\mathit{invariant}\ (\mathit{prog}\ S) \\
&(\!|\mathit{calls} = \mathit{calls}\ S{\restriction}_{\mathit{committedCalls}\ S}, \\
&\quad \mathit{happensBefore} = \mathit{happensBefore}\ S\ |r\ \mathit{committedCalls}\ S, \\
&\quad \mathit{callOrigin} = \mathit{callOrigin}\ S{\restriction}_{\mathit{committedCalls}\ S}, \\
&\quad \mathit{txOrigin} = \mathit{txOrigin}\ S{\restriction}_{\mathit{committedTransactions}\ S}, \mathit{knownIds} = \mathit{knownIds}\ S, \\
&\quad \mathit{invocOp} = \mathit{invocOp}\ S, \mathit{invocRes} = \mathit{invocRes}\ S|\!)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{committedCalls}\ S = \\
&\{c \mid \exists\, tx.\ \mathit{callOrigin}\ S\ c \triangleq tx \wedge \mathit{txStatus}\ S\ tx \triangleq \mathit{Committed}\}
\end{aligned}
$$

Expressiveness of Invariants As mentioned above, specifications in our proof framework are by design limited to invariants about so-called invariant contexts. The invariant context does not include parts of the state that are local to individual procedure invocations. Thus, certain functional properties cannot be expressed as invariants in our framework, while they can be expressed as properties in Isabelle/HOL when directly using the interleaving semantics.

One class of properties not expressible with our invariants are liveness properties, i.e. statements about infinite executions [AS85]. Neither can probabilistic properties like fairness or properties about the running time and resource consumption of programs be expressed.

However, we can express many temporal properties as invariants can access the history and relation between procedure invocations and database operations. This makes out specifications more expressive than classical invariants on states, where such properties can only be defined using ghost variables to record the necessary history information.

The restriction on global states is not a restriction for the expressiveness of specifications, as it does not concern the externally observable behavior. The externally observable behavior is available in the history of procedure invocations, although some aspects as timing and total order of procedure invocations is not available in specifications.

For verification however, the question is, whether it is possible to formulate sufficiently strong invariants such that all verification steps can be completed. Here, the limitation on invariant contexts might be a problem, which we need to address when developing our proof technique in the following sections.

One known limitation concerns reasoning about unique identifiers. Information about the generated unique identifiers is not part of the invariant context. However, there are some implicit dependencies between different procedure invocations due to the uniqueness constraint. This cannot be expressed with our invariants. However, in most cases we can still reason about programs with unique identifiers through the generic properties about well-formed programs that we discuss in Section 6.3.

5.1.1. Correct Programs

Using the invariant checks in our transition relation $S \xrightarrow{tr}{}^* S'$, we can define program correctness as follows:

$$traces\ program \equiv \{tr \mid \exists\, S'.\ initialState\ program \xrightarrow{tr}{}^*\ S'\}$$

$$actionCorrect\ a \equiv a \neq AInvcheck\ False \wedge a \neq ALocal\ False$$

$$traceCorrect\ trace \equiv \forall\, (i,\ a) \in trace.\ actionCorrect\ a$$

$$programCorrect\ program \equiv \forall\, trace \in traces\ program.\ traceCorrect\ trace$$

The set *traces* includes all traces admitted by a program; the predicate *traceCorrect* defines that a trace is correct if it only contains correct actions. There are two cases of actions we consider as incorrect: 1. Invariant checks with result *False* 2. Local steps with check *False*. Using the definition of correct traces, we define a program to be correct if all its traces are correct.

With these definitions we are in principle ready to verify the correctness of applications in Isabelle. The naive approach of using the definition with an induction over the possible steps is however not very practical. We have to consider all possible traces, which includes the interleavings of several concurrent procedure invocations that are allowed by the transition relation. Thus, the invariant required for the induction would need to characterize every single program state of a procedure invocation, leading to invariants which are at least linear in the number of steps in a procedure.

5.2. Reduction to Single-invocation Semantics

To address the challenge of handling concurrency in our setting, we have developed a proof technique, which reduces the formal verification problem to a simpler problem, where we can reason about only one procedure invocation at a time.

Our proof technique is based on invariants and reduces the proof obligations to checking that the initial system state satisfies the invariant and that each procedure invocation maintains the invariant. When verifying a single procedure invocation, the effects of other invocations only need to be considered at specific program points, namely at the procedure invocation and before the start of transactions. We use the invariant and generic properties of executions to reason about possible state changes at these program points. For the procedure to be verified, we must then guarantee that the invariant is maintained at the end of transactions, right after the start of a procedure invocation, and after returning from a procedure invocation. The latter two are necessary because at these program points the information stored in the history of procedure invocations is updated.

Technically, we formalize the reduction using a second operational semantics, the *single-invocation semantics*, which we present in this section. In Section 5.3, we then prove that reduction from the interleaving semantics to the

single-invocation semantics is sound: If a program is correct with respect to the single-invocation semantics, it is also correct in the interleaving semantics.

The main difference between the two semantics is that the single-invocation semantics only allows steps in a single invocation. Effects from different invocations are reflected by nondeterministic steps in the rules for starting a procedure invocation and beginning a transaction. In these cases, the rules of the single-invocation semantics allow an arbitrary state change, assuming that the invariant is maintained, the new state is well-formed, and the history of the new state is an extension of the former history.

Moreover, the single-invocation semantics does not include a dedicated step to check the invariant. The invariant has to be checked in the following three steps: directly after a procedure invocation (*S-invocation*), after the end of a transaction (*S-commit*), and after a procedure invocation returns to the client (*S-return*).

Correspondingly, we adapted the transition relation to be $S \xrightarrow{i,a,v} S'$ for a single step. The value v is *True* if the step fulfills the necessary invariant checks. In the following we will often abbreviate the pair (a, v) using a single variable and write $S \xrightarrow{i,a} S'$, where a contains both the action and the correctness indicator v (c.f. Isabelle's handling of tuples introduced in Section 3.4.1).

We again define $S \xrightarrow{tr}* S'$ as the transitive closure. However, all steps must be on the same invocation now and the traces only include the action a and the value v. Formally, the construction of the transitive closure and the trace is described by the two rules below:

1. $S \xrightarrow{(i,\ [])}* S$
2. $S \xrightarrow{(i,\ tr)}* S' \wedge S' \xrightarrow{(i,\ a)} S'' \Longrightarrow S \xrightarrow{(i,\ tr\ @\ [a])}* S''$

A program is *correct* with respect to the single-invocation semantics if for all possible executions the trace contains only actions for which v is true. Thus, the correctness of a program with respect to the single-invocation semantics can be formalized as:

$$traceCorrect\text{-}s\ trace \equiv \forall\, a.\ (a,\ False) \notin trace$$

$$programCorrect\text{-}s\ program \equiv$$
$$\forall\, trace\ i\ S.\ initialState\ program \xrightarrow{(i,\ trace)}* S \longrightarrow traceCorrect\text{-}s\ trace$$

Figures 5.5 and 5.6 show the rules of the single-invocation semantics. The rules *local*, *DB-operation*, and *new-id* are almost identical to the rules of the interleaving semantics, so we do not elaborate on them here. The interesting differences between the two semantics are in the handling of invocations and transactions. We explain these differences below.

s-invocation:

$$
\begin{array}{l}
\mathit{invocOp}\ S\ i = \mathit{None} \wedge \\
\mathit{procedure}\ (\mathit{prog}\ S)\ \mathit{proc} = (\mathit{initState},\ \mathit{impl}) \wedge \\
\mathit{uniqueIds}\ \mathit{proc} \subseteq \mathit{knownIds}\ S' \wedge \\
\mathit{state\text{-}wellFormed}\ S' \wedge \\
(\forall\, \mathit{tx}.\ \mathit{txStatus}\ S'\ \mathit{tx} \neq \mathit{Some}\ \mathit{Uncommitted}) \wedge \\
\mathit{invariant\text{-}all}\ S' \wedge \\
\mathit{invocOp}\ S'\ i = \mathit{None} \wedge \\
\mathit{prog}\ S' = \mathit{prog}\ S \wedge \\
S'' = S' \\
(\!|\mathit{localState} := \mathit{localState}\ S'(i \mapsto \mathit{initState}), \\
\quad \mathit{currentProc} := \mathit{currentProc}\ S'(i \mapsto \mathit{impl}), \\
\quad \mathit{visibleCalls} := \mathit{visibleCalls}\ S'(i \mapsto \varnothing), \\
\quad \mathit{invocOp} := \mathit{invocOp}\ S'(i \mapsto \mathit{proc})|\!) \wedge \\
\mathit{valid} = \mathit{invariant\text{-}all}\ S'' \wedge (\forall\, \mathit{tx}.\ \mathit{txOrigin}\ S''\ \mathit{tx} \neq \mathit{Some}\ i) \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{AInvoc}\ \mathit{proc},\ \mathit{valid})}_{\cdot} S''
\end{array}
$$

s-return:

$$
\begin{array}{l}
\mathit{localState}\ S\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S\ i \triangleq f \wedge \\
f\ \mathit{ls} = \mathit{Return}\ \mathit{res} \wedge \\
\mathit{currentTx}\ S\ i = \mathit{None} \wedge \\
S' = S \\
(\!|\mathit{localState} := (\mathit{localState}\ S)(i := \mathit{None}), \\
\quad \mathit{currentProc} := (\mathit{currentProc}\ S)(i := \mathit{None}), \\
\quad \mathit{visibleCalls} := (\mathit{visibleCalls}\ S)(i := \mathit{None}), \\
\quad \mathit{invocRes} := \mathit{invocRes}\ S(i \mapsto \mathit{res}), \\
\quad \mathit{knownIds} := \mathit{knownIds}\ S \cup \mathit{uniqueIds}\ \mathit{res}|\!) \wedge \\
\mathit{valid} = \mathit{invariant\text{-}all}\ S' \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{AReturn}\ \mathit{res},\ \mathit{valid})}_{\cdot} S'
\end{array}
$$

s-local:

$$
\begin{array}{l}
\mathit{localState}\ S\ i \triangleq \mathit{ls} \wedge \mathit{currentProc}\ S\ i \triangleq f \wedge f\ \mathit{ls} = \mathit{LocalStep}\ \mathit{ok}\ \mathit{ls}' \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{ALocal}\ \mathit{ok},\ \mathit{ok})}_{\cdot} S(\!|\mathit{localState} := \mathit{localState}\ S(i \mapsto \mathit{ls}')|\!)
\end{array}
$$

s-newId:

$$
\begin{array}{l}
\mathit{localState}\ S\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S\ i \triangleq f \wedge \\
f\ \mathit{ls} = \mathit{NewId}\ \mathit{ls}' \wedge \\
\mathit{generatedIds}\ S\ \mathit{uid} = \mathit{None} \wedge \\
\mathit{uniqueIds}\ \mathit{uidv} = \{\mathit{uid}\} \wedge \mathit{ls}'\ \mathit{uidv} \triangleq \mathit{ls}'' \wedge \mathit{uid} = \mathit{to\text{-}nat}\ \mathit{uidv} \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{ANewId}\ \mathit{uidv},\ \mathit{True})}_{\cdot} S(\!|\mathit{localState} := \mathit{localState}\ S(i \mapsto \mathit{ls}''), \\
\qquad\qquad\qquad\qquad \mathit{generatedIds} := \mathit{generatedIds}\ S(\mathit{uid} \mapsto i)|\!)
\end{array}
$$

Figure 5.5.: *Single-invocation semantics (Part 1).*

s-beginAtomic:

$$
\begin{array}{l}
\mathit{localState}\ S\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S\ i \triangleq f \wedge \\
f\ \mathit{ls} = \mathit{BeginAtomic}\ \mathit{ls}' \wedge \\
\mathit{currentTx}\ S\ i = \mathit{None} \wedge \\
\mathit{txStatus}\ S\ t = \mathit{None} \wedge \\
\mathit{prog}\ S' = \mathit{prog}\ S \wedge \\
\mathit{state\text{-}monotonicGrowth}\ i\ S\ S' \wedge \\
\mathit{invariant\text{-}all}\ S' \wedge \\
(\forall\ \mathit{tx}.\ \mathit{txStatus}\ S'\ \mathit{tx} \neq \mathit{Some}\ \mathit{Uncommitted}) \wedge \\
\mathit{state\text{-}wellFormed}\ S' \wedge \\
\mathit{state\text{-}wellFormed}\ S'' \wedge \\
\mathit{localState}\ S'\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S'\ i \triangleq f \wedge \\
\mathit{currentTx}\ S'\ i = \mathit{None} \wedge \\
\mathit{visibleCalls}\ S\ i \triangleq \mathit{vis} \wedge \\
\mathit{visibleCalls}\ S'\ i \triangleq \mathit{vis} \wedge \\
\mathit{chooseSnapshot}\ \mathit{vis}'\ \mathit{vis}\ S' \wedge \\
\mathit{consistentSnapshot}\ S'\ \mathit{vis}' \wedge \\
\mathit{txStatus}\ S'\ t = \mathit{None} \wedge \\
(\forall\ c.\ \mathit{callOrigin}\ S'\ c \neq \mathit{Some}\ t) \wedge \\
\mathit{txOrigin}\ S'\ t = \mathit{None} \wedge \\
S'' = S' \\
(\!|\mathit{txStatus} := \mathit{txStatus}\ S'(t \mapsto \mathit{Uncommitted}),\ \mathit{txOrigin} := \mathit{txOrigin}\ S'(t \mapsto i), \\
\quad \mathit{currentTx} := \mathit{currentTx}\ S'(i \mapsto t),\ \mathit{localState} := \mathit{localState}\ S'(i \mapsto \mathit{ls}'), \\
\quad \mathit{visibleCalls} := \mathit{visibleCalls}\ S'(i \mapsto \mathit{vis}')|\!) \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{ABeginAtomic}\ t\ \mathit{vis}',\ \mathit{True})}_{\cdot} S''
\end{array}
$$

s-endAtomic:

$$
\begin{array}{l}
\mathit{localState}\ S\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S\ i \triangleq f \wedge \\
f\ \mathit{ls} = \mathit{EndAtomic}\ \mathit{ls}' \wedge \\
\mathit{currentTx}\ S\ i \triangleq t \wedge \\
S' = S \\
(\!|\mathit{localState} := \mathit{localState}\ S(i \mapsto \mathit{ls}'),\ \mathit{currentTx} := (\mathit{currentTx}\ S)(i := \mathit{None}), \\
\quad \mathit{txStatus} := \mathit{txStatus}\ S(t \mapsto \mathit{Committed})|\!) \wedge \\
\mathit{state\text{-}wellFormed}\ S' \wedge \mathit{valid} = \mathit{invariant\text{-}all}\ S' \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{AEndAtomic},\ \mathit{valid})}_{\cdot} S'
\end{array}
$$

s-dbop:

$$
\begin{array}{l}
\mathit{localState}\ S\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S\ i \triangleq f \wedge \\
f\ \mathit{ls} = \mathit{DbOperation}\ \mathit{Op}\ \mathit{ls}' \wedge \\
\mathit{currentTx}\ S\ i \triangleq t \wedge \\
\mathit{calls}\ S\ c = \mathit{None} \wedge \\
\mathit{querySpec}\ (\mathit{prog}\ S)\ \mathit{Op}\ (\mathit{getContext}\ S\ i)\ \mathit{res} \wedge \mathit{visibleCalls}\ S\ i \triangleq \mathit{vis} \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{ADbOp}\ c\ \mathit{Op}\ \mathit{res},\ \mathit{True})}_{\cdot} S(\!|\mathit{localState} := \mathit{localState}\ S(i \mapsto \mathit{ls}'\ \mathit{res}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{calls} := \mathit{calls}\ S(c \mapsto \mathit{Call}\ \mathit{Op}\ \mathit{res}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{callOrigin} := \mathit{callOrigin}\ S(c \mapsto t), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{visibleCalls} := \mathit{visibleCalls}\ S(i \mapsto \\
\mathit{vis} \cup \{c\}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{happensBefore} := \\
\mathit{happensBefore}\ S \cup \mathit{vis} \times \{c\}|\!)
\end{array}
$$

Figure 5.6.: *Single-invocation semantics (Part 2).*

Rule S-invocation. An invocation can only be executed at the beginning of the trace, since the rule demands that the invocation i is not yet used in the current state S. The rule then nondeterministically chooses a state S' which satisfies the invariant and starts the procedure invocation, which yields state S''. The rule also allows us to assume that there are no uncommitted transactions at the start of an invocation and that we start from a well-formed state. A state is defined to be well-formed if it is reachable from the initial state.

We then check whether the invariant holds in S'' and record the result in the trace. This is necessary, because invariants can refer to unfinished procedure invocations, so starting an invocation can cause an invariant violation.

Rule S-return. For a return statement, we check the invariant in the state after completing the invocation.

Rule S-beginAtomic At the beginning of a transaction a new snapshot is determined, which means that this is a place where changes from concurrent invocations might become visible to the current invocation. We model this with a nondeterministic state change from the current state S to a new state S'. The state S' is restricted by the invariant and the predicate $\mathit{state\text{-}monotonicGrowth}(i, S, S')$. Moreover, we can assume that there are no uncommitted transactions in state S'.

The idea of the *state-monotonicGrowth* predicate is to capture generic properties that can be derived from the fact that the history grows monotonically and past events cannot change. Formally, we define it to mean that state S' is reachable from state S with steps on invocations other than i and without any crashes:

$$\begin{array}{l}\mathit{state\text{-}monotonicGrowth}\ i\ S\ S' \equiv \\ \mathit{state\text{-}wellFormed}\ S \wedge \\ (\exists\, tr.\ S \xrightarrow{tr}{}^{*}\ S' \wedge (\forall\, (i',\ a) \in tr.\ i' \neq i) \wedge (\forall\, i'.\ (i',\ \mathit{ACrash}) \notin tr))\end{array}$$

This definition allows us to use any general property we can prove about steps taken in other invocations.

The remaining aspects of the *s-beginAtomic* rule describe the other state changes and are equivalent to the interleaving semantics, or they are additional invariants that we make available in the rule to help with verification.

Rule S-endAtomic When a transaction is committed, we check the invariant in the state after the commit and record the result of the invariant check in the trace. This ensures that an execution is considered incorrect if a transaction breaks the invariant.

5.3. Formalized Soundness Proof

We now show that it is in fact sufficient to prove a program correct with respect to the single-invocation semantics in order to ensure correctness in all

possible concurrent executions according to the interleaving semantics.

We start with a high level overview of the proof and then dive into the details in the subsections of this chapter. Figure 5.7 illustrates the main steps in our proof with an example, which we will pick up in the definitions below.

The first step is to limit the possible interleaving of invocations in a trace. We show that for verification, it is sufficient to consider only traces where all context switches from one invocation to another occur at specific actions. We call such traces *packed*.

Definition 5.3.0.1 (Allowed Context Switches) Only invocations and the start of a transaction are allowed context switches:

$$\begin{aligned}&\textit{allowed-context-switch action} \equiv \\ &(\exists\, \textit{txId txns}.\ \textit{action} = \textit{ABeginAtomic txId txns}) \vee (\exists\, p.\ \textit{action} = \textit{AInvoc } p)\end{aligned}$$

Definition 5.3.0.2 (Packed traces) A trace tr is **packed** for a procedure invocation i if it only switches to invocation i with an action that is an allowed context switch:

$$\begin{aligned}&\textit{packed-trace-i } tr\ i \equiv \\ &\forall\, k.\ 0 < k \wedge k < |tr| \wedge \textit{get-invoc } tr_{[k]} = i \wedge \textit{get-invoc } tr_{[k-1]} \neq i \longrightarrow \\ &\quad \textit{allowed-context-switch } (\textit{get-action } tr_{[k]})\end{aligned}$$

We say a trace is **packed** if it is packed for all procedure invocations:

$$\textit{packed-trace } tr = (\forall\, i.\ \textit{packed-trace-i } tr\ i)$$

Lemma 5.3.0.3 (Reduction to packed traces) A program is correct if all its traces that are packed and do not contain *crash*-steps are correct.
(The formalization and extended proof are given in 5.3.4.2 on page 73)

Proof (sketch) The proof is essentially a reduction argument [Lip75] which uses commutativity of actions performed on different invocations.

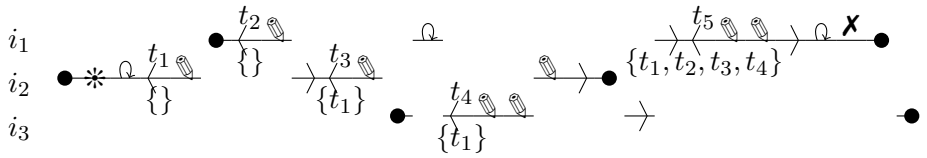
Let tr be a failing trace of the program, i.e. a trace containing a failing invariant check. We show that we can then construct a packed trace which is also failing. We consider the prefix of tr up to the first failing invariant check. We then can reorder the actions in this prefix to get a packed trace.

We prove this by induction over the number of procedure invocations, which are not yet packed. To show that a single invocation i can be packed without unpacking any other invocation, we use another induction over the minimal index k with a not-allowed invocation switch. This cannot be the first action for invocation i, since *invoc* is an allowed invocation switch. Thus, let k' be the last action on invocation i before k. We then split the trace into $tr = tr_{[..k'-1]} \cdot tr_{k'} \cdot tr_{[k'+1..k-1]} \cdot tr_k \cdot tr_{[k+1..]}$ and reorder it by moving tr_k to the front to get $tr' = tr_{[..k'-1]} \cdot tr_{k'} \cdot tr_k \cdot tr_{[k'+1..k-1]} \cdot tr_{[k+1..]}$. By changing the order of the trace, we have eliminated all unwanted invocation switches up to index k. We can show that the action of tr_k commutes with the actions it is swapped with, which are all from a different invocation.

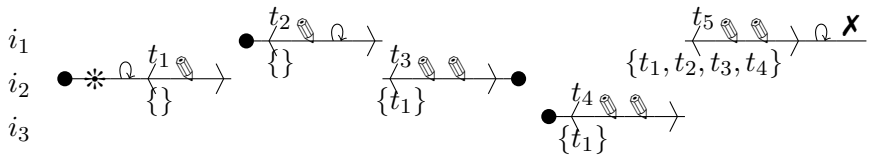
Trace actions: ● begin a procedure invocation, ● return from an invocation,

start a transaction, commit a transaction, database operation,
create a new unique identifier, local steps, and ✗ a failing invariant.
We annotate each start of a transaction with a transaction id and the set of transactions that are visible to this transaction.

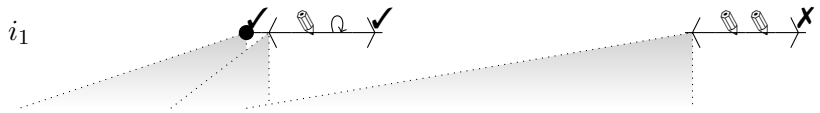
Step 1: Assume we have a failing trace in the interleaving semantics, for example:



Step 2: In Lemma 5.3.0.3, we show that in this case there is an equivalent packed trace that is also failing. We construct this packed trace by moving actions to the front. This reordering does not invalidate any snapshots and since snapshots are fixed in the trace, the effect of the trace is preserved. For the example above, we can construct the following packed trace which ends with a failing invariant:



Step 3: In Lemma 5.3.0.4, we show that there is a corresponding trace in the single invocation semantics, where we only consider procedure invocation i_1 while actions from other invocations are summarized using invariants (visualized by "lightcones" in the dotted lines). The proof obligation for the single invocation semantics is to check the invariant after an invocation return and after each transaction commit. In the picture below, the proof obligation is always to check that the actions between a light source and a check mark preserve the invariant.



Thus, if we prove that no invariant check ever fails in the single invocation semantics, we have shown the correctness of the application.

Figure 5.7.: *Overview of our soundness proof applied to an exemplary trace.*

The commutativity proof involves a case distinction over all possible actions in the system. An interesting case here is moving an *endAtomic* action to the front. In principle, this could affect other transactions, but since we do not change the transaction snapshots when reordering the trace, we are guaranteed to get the same results. Recall, that this behavior is valid in our semantics of weak consistency, where the chosen snapshot does not need to include all committed transactions. □

Lemma 5.3.0.4 (Simulation) Let tr be a packed trace of an execution starting in state S and ending in state S' where S is well-formed (i.e. reachable from the initial state) and satisfies the invariant and S' does not satisfy the invariant. Moreover, assume that tr is packed and does not contain any crashes. Then, there is an execution with a trace tr' in the single-invocation semantics starting with state S such that tr' is not a correct trace.
(The formalization and extended proof are given in 5.3.7.3 on page 82)

Proof (Proof sketch) We show that the single-invocation semantics can simulate the distributed (interleaving) semantics. To this end, we define a coupling relation between a state S_d of the distributed execution and a state S_i of a single-invocation execution with invocation i. The coupling invariant distinguishes two cases: 1) When the last step in the distributed execution was in invocation i, then the states must be equal. 2) When the last step was on an invocation different from i, it must hold that S_i is greater than S_d with respect to the *growing* relation and that the local state of invocation i is equivalent in S_i and S_d.

In the simulation proof, the case of switching between invocations can only occur at the beginning of invocations or at the start of a transaction (because we assumed the trace is packed). In both cases, the single-invocation semantics allows nondeterministic state-transitions, which enable the single-invocation execution to catch up with the distributed invocation. The remaining cases are straight-forward. □

Theorem 5.3.0.5 (Soundness of verification technique) When a program is correct with respect to the single-invocation semantics and the initial state satisfies the invariant, then the program is correct with respect to the interleaving semantics.
(The formalization and extended proof are given in 5.3.7.4 on page 83)

Proof (Proof sketch) We show that all executions are correct. Because of Lemma 5.3.0.3, it is sufficient to consider executions with packed traces without crashes. Let tr be a trace for such an execution.

For the sake of a contradiction assume tr is not a correct trace, i.e. there is a failing invariant check in the trace. We now consider the first time when an invariant-violating state is reached and the prefix of tr leading to this state. As the state does not satisfy the invariant, though the initial state does (by assumption), we can apply Lemma 5.3.0.4 and obtain a failing trace in the single-invocation semantics. However, this is a contradiction to the assumption that the program is correct in the single-invocation semantics. □

This completes the overview of our soundness proof. In the following sections, we will present the steps in the proof in more detail. We first prove some general invariants that are true for all executions in Section 5.3.1. Then we prove the commutativity of certain actions in Section 5.3.2. With this commutativity results we show how we can reorder traces to be packed (Section 5.3.3). We then show that we can exclude crashes for verification (Section 5.3.4), and that we do not need to handle invariant checks in transactions (Section 5.3.5). Finally, we refine the property of packed traces such that context switches between different procedure invocations may only happen at certain actions (Section 5.3.6). With this, we can then prove the soundness of the reduction to the single-invocation semantics in Section 5.3.7.

5.3.1. Execution Invariants

In this subsection, we prove some general properties that hold for each execution of the interleaving semantics. These Lemmata have two purposes:

1. Some properties are required for the proof steps below. In particular, the rules of the single-invocation semantics include additional assumptions, which are not directly expressed in the interleaving rules.
2. The single-invocation semantics allows one to assume the *wellformed* property for states during the execution. The Lemmata in this section allow deriving higher level properties from the *wellformed* predicate, which can be useful when verifying concrete programs. It is also the basis of the predicate abstraction we choose for the Repliss tool (see Section 7.5).

Consistency

We now show that every *wellformed* state satisfies certain consistency guarantees. More precisely, we want to show that the happens-before relation is a strict partial order and that all snapshots are causally and transactionally consistent.

Remember that we defined a snapshot to be a set of database calls. We say that a snapshot is causally consistent if it downwards-closed with respect to the happens-before relation. So, if a call c_1 is in the snapshot and a call c_2 happened before c_1, then c_2 must also be in the snapshot:

$$causallyConsistent\ hb\ vis \equiv \forall\ c1\ c2.\ c1 \in vis \wedge (c2, c1) \in hb \longrightarrow c2 \in vis$$

Transactional consistency of a snapshot can be divided into two cases. The first condition is that all calls in the snapshot must be from committed transactions:

$$transactionConsistent\text{-}committed\ origin\ txSt\ vis \equiv$$
$$\forall\ c\ tx.\ c \in vis \wedge origin\ c \triangleq tx \longrightarrow txSt\ tx \triangleq Committed$$

The second condition is that transactions must be atomic. If a call c_1 is in the snapshot, then all other calls c_2 from the same transaction must also be included:

transactionConsistent-atomic origin vis ≡
∀ *c1 c2*. *c1* ∈ *vis* ∧ *origin c1* = *origin c2* ⟶ *c2* ∈ *vis*

A consistent snapshot is a subset of all database calls in the history that satisfies the three properties introduced above:

consistentSnapshot S vis =
(*vis* ⊆ *dom* (*calls S*) ∧
causallyConsistent (*happensBefore S*) *vis* ∧
transactionConsistent-committed (*callOrigin S*) (*txStatus S*) *vis* ∧
transactionConsistent-atomic (*callOrigin S*) *vis*)

In order to prove that the steps of the interleaving semantics guarantee that all snapshots are consistent, we start by proving that the happens-before relation is always transitive and that snapshots are always causally consistent. We prove both properties by an induction over the steps. Since they depend on each other, we cannot easily split them into separate proofs.

5.3.1.1 lemma *wellFormed-state-causality*:
assumes *wf*: *state-wellFormed S*
shows ⋀*s vis*. *visibleCalls S s* ≜ *vis* ⟶ *causallyConsistent* (*happensBefore S*) *vis*
and *trans* (*happensBefore S*)

Next, we show that snapshots are always transactional consistent.

5.3.1.2 lemma *wellFormed-state-transaction-consistent*:
assumes *wf*: *state-wellFormed S*
— contains only committed calls and calls from current transaction:
shows ⋀*s vis c tx*. ⟦*visibleCalls S s* ≜ *vis*; *c*∈*vis*; *callOrigin S c* ≜ *tx*⟧ ⟹ *txStatus S tx* ≜ *Committed* ∨ *currentTx S s* ≜ *tx*
— contains all calls from a transaction
and ⋀*s vis c c′*. ⟦*visibleCalls S s* ≜ *vis*; *c*∈*vis*; *callOrigin S c* = *callOrigin S c′*⟧ ⟹ *c′*∈*vis*
— happens-before consistent with transactions
and ⋀*x y x′ y′*. ⟦*callOrigin S x* ≠ *callOrigin S y*; *callOrigin S x* = *callOrigin S x′*; *callOrigin S y* = *callOrigin S y′* ⟧ ⟹ (*x*,*y*) ∈ *happensBefore S* ⟷ (*x′*, *y′*) ∈ *happensBefore S*
— happens-before only towards committed transactions or to the same transaction
and ⋀*x y tx tx′*. ⟦(*x*,*y*)∈*happensBefore S*; *callOrigin S y* ≜ *tx*; *callOrigin S x* ≜ *tx′*⟧ ⟹ *txStatus S tx′* ≜ *Committed* ∨ *tx′* = *tx*

We again prove this by induction over the step and need a few additional properties for the induction to succeed. Besides the two properties for transactional consistency that we defined above, we also show that the happens-before relation is consistent with transactions and that calls cannot happen before calls from other transactions that are not yet committed.

Finally, we can combine the results above to show that all snapshots are consistent when not inside of a transaction:

5.3.1.3 lemma *wellFormed-state-consistent-snapshot*:
assumes *wf*: *state-wellFormed S*
assumes *vis*: *visibleCalls S s* ≜ *vis*
assumes *noTx*: ⋀*c tx*. *currentTx S s* ≜ *tx* ⟹ *callOrigin S c* ≠ *Some tx*
shows *consistentSnapshot S vis*

Further Properties

In the following we list some additional properties about well-formed states, which we will need in later sections. All of these can be proven with a simple induction over the steps in the system, so we omit details about the proofs below.

5.3.1.4 lemma *wf-no-invocation-no-origin*:
assumes *state-wellFormed S*
and *invocOp S i = None*
shows *txOrigin S tx ≠ Some i*

By induction over steps.

5.3.1.5 lemma *wf-no-txStatus-origin-for-nothing*:
assumes *wf*: *state-wellFormed S*
and *txStatusNone*: *txStatus S tx = None*
shows *callOrigin S c ≠ Some tx*

By induction over steps.

5.3.1.6 lemma *wellFormed-currentTx-unique-h*:
assumes *a1*: *state-wellFormed S*
shows ∀ *sa sb t. currentTx S sa ≜ t ⟶ currentTx S sb ≜ t ⟶ sa = sb*
and ∀ *sa t. currentTx S sa ≜ t ⟶ txStatus S t ≜ Uncommitted*

By induction over steps.

5.3.1.7 lemma *wellFormed-visibleCallsSubsetCalls-h*:
assumes *a1*: *state-wellFormed S*
shows *happensBefore S ⊆ dom* (*calls S*) × *dom* (*calls S*)
and ⋀*vis s. visibleCalls S s ≜ vis ⟹ vis ⊆ dom* (*calls S*)

By induction over steps.

5.3.1.8 lemma *wellFormed-visibleCallsSubsetCalls2*:
assumes ‹*state-wellFormed S*›
and ‹*visibleCalls S sb ≜ visa*›
and ‹*calls S c = None*›
shows ‹*c∉visa*›

This is a direct consequence of Lemma 5.3.1.7.

5.3.2. Commutativity

The basis of the soundness proof is the commutativity of actions. This allows us to reorder actions in an interleaved trace, such that transactions are grouped together and context switches only appear at certain points in the trace.

We define a predicate *canSwap* for two actions a and b. The predicate states that action b can be swapped with a and executed before it without changing the resulting state, given that both actions are executed on different invocations and the initial state is well-formed:

$$
\begin{aligned}
&\mathit{canSwap}\ t\ a\ b \equiv \\
&\forall\ \mathit{C1}\ \mathit{C2}\ \mathit{i1}\ \mathit{i2}. \\
&\quad \mathit{i1} \neq \mathit{i2} \wedge \mathit{C1} \xrightarrow{[(\mathit{i1},\ a),\ (\mathit{i2},\ b)]}{}^{*}\ \mathit{C2} \wedge \mathit{state\text{-}wellFormed}\ \mathit{C1} \longrightarrow \\
&\quad \mathit{C1} \xrightarrow{[(\mathit{i2},\ b),\ (\mathit{i1},\ a)]}{}^{*}\ \mathit{C2}
\end{aligned}
$$

For technical reasons, the definition also requires the parameter t, which is the type of local states in $C1$ and $C2$.

We can then show that this predicate holds for most action pairs:

Lemma 5.3.2.1 (CanSwap Cases)

lemma *canSwap-cases*:
  assumes *no-begin-atomic*: $\bigwedge$*txId txns. b* $\neq$ *ABeginAtomic txId txns*
    and *no-invoc*: $\bigwedge$*p. b* $\neq$ *AInvoc p*
    and *no-invcheck-a*: $\neg$*is-AInvcheck a*
    and *no-invcheck-b*: $\neg$*is-AInvcheck b*
    and *no-fail-a*: *a* $\neq$ *ACrash*
    and *no-fail-b*: *b* $\neq$ *ACrash*
shows *canSwap t a b*

The proof is done by case distinction over a and b. Most cases can be solved automatically by unfolding the definitions. Below, we only present the interesting cases which are the ones requiring the well-formedness of the initial state.

1. Case $(i_1, \mathit{beginAtomic}(tx, txns)) \cdot (i_2, \mathit{endAtomic})$:

   In this case the swap would fail, if the current transaction in i_2 would be equal to tx. Then the *endAtomic* action would set the *txStatus* to *committed*, which would violate the precondition of *beginAtomic*. However, this situation cannot occur in a *wellformed* state. The precondition of *endAtomic* demands that the current transaction is tx and with well-formedness (Lemma 5.3.1.6) we get that the *txStatus* is *uncommitted*. Thus the *beginAtomic* cannot use the same transaction.

   Moreover, moving the *endAtomic* to the front cannot changed the snapshot induced by the *txns* in *beginAtomic*, since the new transaction cannot come before any of the already committed transaction in the *txns* set.

2. Case $(i_1, \mathit{beginAtomic}(tx, txns)) \cdot (i_2, \mathit{dbOp}(c, op, r))$:

   The *beginAtomic* action determines the snapshot of the new transaction. We have to show that this does not change if the database operation in i_2 is executed earlier. First, the call c from i_2 cannot be included in the snapshot, since the precondition of *dbOp* requires it to be a fresh call. Secondly, after swapping the two actions, the call c cannot be included in the snapshot since it cannot have happened before any of the transactions in *txns*.

3. Case $(i_1, dbOp(c_1, op_1, r_1)) \cdot (i_2, dbOp(c_2, op_2, r_2))$:

   Here, we can show that the operation contexts are equivalent after changing the order. This is easy to show using well-formedness properties (Lemma 5.3.1.8), since neither c_1 nor c_2 can be in the set of visible calls in the initial state.

The definition of *canSwap* only allows swapping of a single action-pair in the trace. We now show that this can be generalized to swap an action in the trace with a whole subsequence of the trace.

Lemma 5.3.2.2 (CanSwap Sequence)
Let $tr \cdot (i, a)$ be a trace, where no action in tr is from invocation i, every action in tr can be swapped with a, and where tr contains no crashes. We can then move the last action (i, a) to the front resulting in trace $(i, a) \cdot tr$, which results in the same state when started from a *wellformed* initial state.

lemma *swapMany*:
  fixes *C1* :: (*'proc*::*valueType*, *'ls*, *'op*, *'any*::*valueType*) *state*
    and *t* :: *'ls itself*
  assumes *steps*: $C1 \xrightarrow{tr \ @ \ [(i,a)]}{}^{*} C2$
    and *tr-different-session*: $\bigwedge x.\ x \in set\ tr \Longrightarrow get\text{-}invoc\ x \neq i$
    and *tr-canSwap*: $\bigwedge x.\ x \in set\ tr \Longrightarrow canSwap\ t\ (get\text{-}action\ x)\ a$
    and *wf*: *state-wellFormed C1*
    and *noFail*: $\bigwedge i.\ (i, ACrash) \notin set\ tr$
shows $C1 \xrightarrow{[(i,a)] \ @ \ tr}{}^{*} C2$

The proof is by reverse induction over tr. When tr is the empty sequence, the statement is trivial. Otherwise, we can swap the last action in tr with (i, a). This is possible since by assumption all actions in tr are swappable with action a, are on a different invocation and because the state is *wellformed*. To prove well-formedness, we use the fact that the initial state is well-formed and that the trace does not contain crashes. Then we can apply the induction hypothesis to complete the proof.

5.3.3. Packed Traces

A packed trace is a trace with limited interleaving: In a packed trace a context switch from one procedure invocation to another is only allowed at a *beginAtomic* or *invoc* action. We formalize this with the following definitions:

$$allowed\text{-}context\text{-}switch(action) \equiv (\exists txId, txns.\ \ action = beginAtomic(txId, txns)) \\ \vee (\exists p.\ \ action = invoc(p))$$

$$packed\text{-}trace(tr) \equiv \forall i.\ 0 < i < length(tr) \wedge get\text{-}invoc(tr_{i-1}) \neq get\text{-}invoc(tr_i) \\ \longrightarrow allowed\text{-}context\text{-}switch(get\text{-}action(tr_i))$$

In this section, we show that it is sufficient to consider packed traces when reasoning about application correctness: If there is an incorrect trace with no crashes, then there also is an incorrect *and* packed trace with no crashes.

Lemma 5.3.3.1 (Packed Traces)

lemma *pack-incorrect-trace*:
  assumes *steps*: *initialState program* $\xrightarrow{tr}{}^{*}$ *C*
    and *noFail*: $\bigwedge s.$ $(s, ACrash) \notin set\ tr$
    and *notCorrect*: $\neg traceCorrect\ tr$
  shows $\exists\, tr'\ C'.$ *packed-trace tr′*
      $\wedge$ (*initialState program* $\xrightarrow{tr'}{}^{*}$ *C′*)
      $\wedge$ $(\forall s.\ (s, ACrash) \notin set\ tr')$
      $\wedge$ $\neg traceCorrect\ tr'$

For the proof we proceed in two steps. We first show that we can transform the trace to be packed for a single invocation, then we repeat this step to show that we can have a packed trace for all invocations.

We say that a trace is packed for an invocation i, if all context switches in the trace from another invocation to i occur at an action with *allowed-context-switch* (i.e. the start of invocation i or the start of a new transaction).

packed-trace-i tr i ≡
$\forall k.\ 0 < k \wedge k < |tr| \wedge$ *get-invoc* $tr_{[k]} = i \wedge$ *get-invoc* $tr_{[k-1]} \neq i \longrightarrow$
  allowed-context-switch (*get-action* $tr_{[k]}$)

Obviously, a trace is packed if it is packed for all invocations:

packed-trace tr = $(\forall i.$ *packed-trace-i tr i*)

We now prove the Lemma that allows us to transform a single invocation into a packed invocation. The transformed trace also maintains some important properties of the original trace. All previously packed invocations are still packed and the new trace also dos not contain any crashes or invariant checks. This allows us to repeatedly apply this Lemma to pack all invocations.

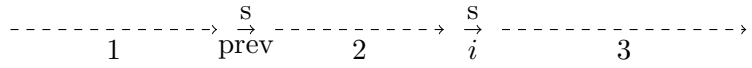
Lemma 5.3.3.2 (Pack trace for one invocation)

lemma *pack-trace-for-one-session*:
  assumes *steps*: *initialState program* $\xrightarrow{tr1}{}^{*}$ *C*
    and *noFail*: $\bigwedge s.$ $(s, ACrash) \notin set\ tr1$ (**is** $\bigwedge s.$ - $\notin$ *set ?tr*)
    and *noInvcheck*: $\bigwedge s\ a.$ $(s, a) \in set\ tr1 \Longrightarrow \neg is\text{-}AInvcheck\ a$
  shows $\exists\, tr'.$ *packed-trace-i tr′ s*
      $\wedge$ (*initialState program* $\xrightarrow{tr'}{}^{*}$ *C*)
      $\wedge$ $(\forall s.$ *packed-trace-i tr1 s* $\longrightarrow$ *packed-trace-i tr′ s*)
      $\wedge$ (*set tr′* = *set tr1*)

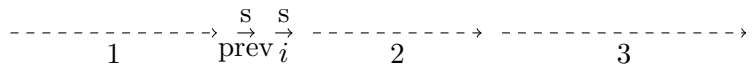
Proof Assume the trace is not yet packed for invocation s and let i be the smallest index with a problematic context switch. We show that we can construct a trace with no problematic context switch up to index $i + 1$, while

maintaining the other properties of the trace relevant for the Lemma. By repeating this step, we can construct a trace that is packed for invocation s.

We assumed that the action at i is an action in invocation s which is not an allowed context switch. This implies that the action is not the start of invocation s and thus there must be a previous action in invocation s in the trace. Let *prev* be the last action in s before i. We can now split the trace into 5 parts:



We then consider the trace tr', where action i is moved to the front, directly following *prev*:



Now, in tr' we only have allowed context switches on invocation s up to and including index i. It remains to show that tr' maintains all the relevant properties requires for the Lemma.

1. As we only changed the order of the trace and not the contents, tr' does not contain any crashes or invariant checks.

2. If an invocation was already packed in tr, it is still packed in tr'. The only new context switch newly introduced in tr' can be at the beginning of part 3 of the trace. If this action is from a session that is packed in tr, then it must already have been an allowed context switch, since we were coming from invocation s. So the corresponding invocation is still packed.

3. The execution of trace tr' ends in the same state as the execution of tr. For this we use Lemma 5.3.2.2 from our commutativity results. Since the action at i is no allowed context switch by assumption and the trace contains no invariant checks or crashes, all actions in part 2 of the trace satisfy the *canSwap* predicate with respect to the action at position i (see Lemma 5.3.2.1).

5.3.4. No Crashes

We now show that it is sufficient to consider executions without crashes when verifying a program. The intuition behind this is that it is not possible to distinguish a crashed invocation from an invocation that executes no further steps.

Lemma 5.3.4.1 (Can Ignore Crashes) All traces of a program are correct, if and only if all traces without crashes are correct.

lemma *can-ignore-fails*:
shows $(\forall\, tr{\in}traces\ program.\ traceCorrect\ tr)$
$\longleftrightarrow (\forall\, tr{\in}traces\ program.\ (\nexists\, s.\ (s, ACrash) \in set\ tr) \longrightarrow\ traceCorrect\ tr)$

For the proof, assume we have an incorrect trace tr that contains crashes. We show that the trace tr' where we remove all crashes also is a trace of the program. So for the given trace tr with $S_{init} \xrightarrow{tr}{}^* S'$ we show that there is a state S'' such that $S_{init} \xrightarrow{tr'}{}^* S''$. In the simulation proof, we use the following coupling invariant relating S' and S'': The two states are equivalent except in the invocations i where $(i, crash) \in tr$. For these invocations the fields *localState*, *currentTransaction*, *currentProc*, and *visibleCalls* may differ.

The differences in these fields is not relevant, as they are only used in actions on the respective invocations. In particular, they cannot be accessed from invariants. It remains to show that there can be no further action on an invocation after an crash, which follows from the precondition of the respective actions. □

We can combine the results from Lemma 5.3.4.1 with Lemma 5.3.3.1 to get the following Lemma:

Lemma 5.3.4.2 **theorem** *show-programCorrect-noTransactionInterleaving*:
assumes *packedTracesCorrect*:
⋀*trace s*. ⟦
initialState program $\xrightarrow{trace}{}^*$ *s*;
packed-trace trace;
⋀*s*. (*s*, *ACrash*) ∉ *set trace*
⟧ ⟹ *traceCorrect trace*
shows *programCorrect program*

5.3.5. No Invariant Checks

We now show that we do not have to consider traces with invariant checks inside of transactions when verifying a program. Obviously, an invariant check has no effect on the state, so checks where the invariant evaluates to *true* can be removed from a trace. The resulting trace remains its packed property, since the filtering does not remove the allowed context switches. If we combine this with Lemma 5.3.4.2, we get:

Lemma 5.3.5.1

theorem *show-programCorrect-noTransactionInterleaving-no-passing-invchecks*:
assumes *packedTracesCorrect*:
⋀*trace s*. ⟦
initialState program $\xrightarrow{trace}{}^*$ *s*;
packed-trace trace;
⋀*s*. (*s*, *ACrash*) ∉ *set trace*;
⋀*s*. (*s*, *AInvcheck True*) ∉ *set trace*
⟧ ⟹ *traceCorrect trace*
shows *programCorrect program*

Next, we show that we do not have to consider invariant checks inside transactions.

Lemma 5.3.5.2 No Invariant Checks in Transactions

Assume we are given a nonempty trace of the program ending in a failing invariant. Moreover, the trace contains no other invariant checks and no crashes. Then there also is a trace with the aforementioned properties, which additionally does not contain any invariant checks inside a transaction.

lemma *move-invariant-checks-out-of-transactions*:
  assumes *initialState program* $\xrightarrow{trace}{}^{*}$ *S*
    and *packed-trace trace*
    and $\bigwedge s.$ (*s*, *ACrash*) $\notin$ *set trace*
    and $\bigwedge s.$ (*s*, *AInvcheck True*) $\notin$ *set trace*
    and *length trace* > *0*
    and *last trace* = (*s*, *AInvcheck False*)
    and $\bigwedge i\ s'.$ *i*<*length trace* − *1* $\Longrightarrow$ *trace*!*i* $\neq$ (*s'*, *AInvcheck False*)
  shows $\exists$ *trace' s'*.
        ($\exists$ *S'*. *initialState program* $\xrightarrow{trace'}{}^{*}$ *S'*)
      $\wedge$ *packed-trace trace'*
      $\wedge$ ($\forall$ *s*. (*s*, *ACrash*) $\notin$ *set trace'*)
      $\wedge$ ($\forall$ *s*. (*s*, *AInvcheck True*) $\notin$ *set trace'*)
      $\wedge$ (*last trace'* = (*s'*, *AInvcheck False*))
      $\wedge$ *length trace'* > *0*
      $\wedge$ (*no-invariant-checks-in-transaction trace'*)

For the proof, we use induction over the length of the trace. Since the trace only contains a single invariant check by assumption, we only have to consider the case where this invariant check is inside a transaction. In that case we can show that we can move the invariant check one position to the front, obtaining a shorter trace. The invariant check will still fail since local actions in a transaction cannot have any influence on invariants.

Using this result, we can again limit the kind of traces that have to be considered to verify that a program is correct:

Lemma 5.3.5.3

theorem *show-programCorrect-noTransactionInterleaving'*:
  assumes *packedTracesCorrect*:
    $\bigwedge$*trace s*. ⟦
      initialState program $\xrightarrow{trace}{}^{*}$ *s*;
      packed-trace trace;
      $\bigwedge s.$ (*s*, *ACrash*) $\notin$ *set trace*;
      $\bigwedge s.$ (*s*, *AInvcheck True*) $\notin$ *set trace*;
      no-invariant-checks-in-transaction trace⟧
    $\Longrightarrow$ *traceCorrect trace*
shows *programCorrect program*

To prove this we use Lemma 5.3.5.1. Thus, let *tr* be a trace with the properties given in the Lemma. To show that the trace is correct, we use a proof by contradiction. Assume *tr* is not correct and let *i* be the position of the first failing invariant. We then consider only the trace up to and including position *i*. This trace satisfies the preconditions of Lemma 5.3.5.2, so we can obtain a failing trace with no invariants in transactions. However, this contradicts the assumption that all such traces are correct. □

5.3.6. No Context Switches in Transaction

In this section we further refine our argument about *packed* traces. We show we only have to consider traces that have no context switches inside transactions. We formalize this with the predicate *contextSwitchesInTransaction*, which we define as follows:

contextSwitchInTransaction tr i-begin i-switch ≡
∃ *invoc tx txns*.
  i-begin < *i-switch* ∧
  i-switch < |*tr*| ∧
  $tr_{[i\text{-}begin]}$ = (*invoc*, *ABeginAtomic tx txns*) ∧
  (∀ *j*. *i-begin* < *j* ∧ *j* < *i-switch* ⟶ $tr_{[j]} \neq$ (*invoc*, *AEndAtomic*)) ∧
  allowed-context-switch (*get-action* $tr_{[i\text{-}switch]}$)

contextSwitchesInTransaction tr ≡
∃ *i-begin i-switch*. *contextSwitchInTransaction tr i-begin i-switch*

A trace that is *packed* does not automatically fulfill this property. A context switch to another invocation is allowed inside a transaction, although the old invocation would never be allowed to do another step and therefore would not be able to complete the transaction.

Lemma 5.3.6.1 (Remove context switches in transactions)

lemma *remove-context-switches-in-transactions*:
  assumes *a1*: *initialState program* $\xrightarrow{trace}{}^{*}$ *S*
    and *a2*: *packed-trace trace*
    and *a3*: ⋀*s*. (*s*, *ACrash*) ∉ *set trace*
    and *a4*: ⋀*s*. (*s*, *AInvcheck True*) ∉ *set trace*
    and *a5*: *no-invariant-checks-in-transaction trace*
    and *a6*: ¬*traceCorrect trace*
  obtains *trace′ S′*
  where *initialState program* $\xrightarrow{trace'}{}^{*}$ *S′*
    and *packed-trace trace′*
    and ⋀*s*. (*s*, *ACrash*) ∉ *set trace′*
    and ⋀*s*. (*s*, *AInvcheck True*) ∉ *set trace′*
    and *no-invariant-checks-in-transaction trace′*
    and ¬*traceCorrect trace′*
    and ¬*contextSwitchesInTransaction trace′*

We prove this Lemma by an induction over the length of the trace. Without loss of generality, we can assume that the trace is incorrect only because of the last action in the trace (otherwise we can remove everything after the first incorrect action and apply the induction hypothesis).

Next, let i_{switch} be the position of the first context switch in a transaction. We can then remove the action at position $i_{switch} - 1$ from the trace. This yields a shorter trace which maintains the required properties and thus allows us to use the induction hypothesis. As mentioned earlier, it is safe to remove the action as the same invocation can never do another step without violating the *packed* property.

□

With this result we can refine Lemma 5.3.5.3 to also exclude context switches in the transactions that must be considered for correctness checks:

Lemma 5.3.6.2

theorem *show-programCorrect-noTransactionInterleaving''*:
  assumes *packedTracesCorrect*:
    $\bigwedge$*trace s*. [[
      initialState program $\xrightarrow{trace}{}^*$ *s*;
      packed-trace trace;
      ¬*contextSwitchesInTransaction trace*;
      $\bigwedge$*s*. (*s*, *ACrash*) ∉ *set trace*;
      no-invariant-checks-in-transaction trace
    ]] ⟹ *traceCorrect trace*
shows *programCorrect program*

The major consequence of not having context switches in transactions is captured in the following Lemma:

Lemma 5.3.6.3 (At most one current transaction) We show that the only invocation that can contain an active transaction is the invocation that did the last action in the trace, if we have a trace tr starting from a state S and going to a state S', such that:

- The initial state S is *wellformed* and contains no uncommitted transactions.
- The trace is *packed* and contains no context switches in transactions.
- The trace contains no crashes.

lemma *at-most-one-current-tx*:
  assumes *steps*: *S* $\xrightarrow{tr}{}^*$ *S'*
    and *noCtxtSwitchInTx*: ¬*contextSwitchesInTransaction tr*
    and *packed*: *packed-trace tr*
    and *wf*: *state-wellFormed S*
    and *noFails*: $\bigwedge$*s*. (*s*, *ACrash*) ∉ *set tr*
    and *noUncommitted*: $\bigwedge$*tx*. *txStatus S tx* ≠ *Some Uncommitted*
shows ∀ *i*. *currentTx S' i* ≠ *None* ⟶ *i* = *get-invoc* (*last tr*)

The proof is done by induction over the steps in the trace. If the trace is empty, there are no open transactions by assumption. Next, we consider a nonempty trace to which we add an action a.

If action a is not an allowed context switch, then the invocation of the last action in tr must be equal to the invocation of a since the trace is packed by assumption.

On the other hand, if a is an allowed context switch, then there is no active transaction: By the induction hypothesis the only invocation that might have an open transaction is the invocation of the last action in tr. However, we

assumed there are no context switches in transactions, so that case is ruled out by the fact that a is an allowed context switch.

The remainder of a proof is a straight forward case analysis over the possible cases. □

5.3.7. Reduction to Single-invocation Semantics

We now show that the verification problem on the interleaving semantics can be reduced to verification with respect to the single invocation semantics (see Section 5.2). To connect the two semantics we define a coupling relation between states in the one and states in the other. The coupling relation is parameterized by the invocation i on which the single-invocation semantics focuses. Moreover, the coupling relation takes a boolean parameter *sameInvoc*, which is true if the last step in the trace was executed on invocation i. If this is the case, the coupling relation demands that both states are equivalent. Otherwise, the predicate *state-monotonicGrowth* must hold for the two states, meaning that the state in the interleaving semantics might be ahead of the state in the single-invocation semantics. More precisely, *state-monotonicGrowth*(i, S, S') is true if there is a trace tr from S to S' and that trace contains no crashes and no actions on invocation i:

state-coupling S S' i *sameInvoc* ≡
if sameInvoc **then** $S' = S$ **else** *state-monotonicGrowth* i S' S

Using this coupling invariant, we can now show how to transform a trace in the interleaving semantics to a corresponding trace in the single-invocation semantics such that both executions terminate in coupled states. In this first step, we only consider correct executions, which means that the invariant is maintained in all intermediate states of the execution.

Lemma 5.3.7.1 (Convert to single-invocation trace) If we have an execution in the interleaving semantics starting with state satisfying the invariant, then we can convert this trace to a single-invocation trace leading to coupled state. For this, the trace must satisfy the following properties: It must be *packed*, contain no context switches in transactions, and contain no crashes. Moreover, there must not be any uncommitted transactions in the starting state and the program invariant must hold in all states of the trace.

lemma *convert-to-single-session-trace*:
  fixes tr :: ('*proc*::*valueType*, '*op*, '*any*::*valueType*) *trace*
    and i :: *invocId*
    and S S' :: ('*proc*, '*ls*, '*op*, '*any*) *state*
  assumes *steps*: $S \xrightarrow{tr}* S'$
    and *S-wellformed*: *state-wellFormed S*
    and *packed*: *packed-trace tr*
    and *noFails*: $\bigwedge s.\ (s, ACrash) \notin set\ tr$
    and *noUncommitted*: $\bigwedge tx.\ txStatus\ S\ tx \neq Some\ Uncommitted$
    and *noCtxtSwitchInTx*: $\neg contextSwitchesInTransaction\ tr$
    — invariant holds on all states in the execution

and *inv*: $\bigwedge S'\ tr'.\ [\![isPrefix\ tr'\ tr;\ S \xrightarrow{tr'}{}^*\ S']\!] \Longrightarrow invariant\text{-}all\ S'$
and *noAssertionFail*: $\bigwedge a.\ a \in set\ tr \Longrightarrow get\text{-}action\ a \neq ALocal\ False$
shows $\exists\, tr'\ S2.\ (S \xrightarrow{(i,\ tr')}{}^*\ S2)$
$\wedge\ (\forall\ a.\ (a,\ False) \notin set\ tr')$
$\wedge\ (state\text{-}coupling\ S'\ S2\ i\ (tr = [] \vee get\text{-}invoc\ (last\ tr) = i))$

For the proof we use induction over the steps taken in the interleaving semantics. The case for the empty trace is trivial: The coupling relation holds for the state with itself.

For the induction step we consider a nonempty trace with last action a. First, we consider the case where action a is not from invocation i. Here, we make no step in the single-invocation semantics and use the state from the previous step. The coupling invariant is then trivially preserved, since the definition of *state-monotonicGrowth* allows steps to be taken in other invocations.

Now we get to the interesting case where we take a step in the same invocation. We continue using a case distinction on the executed action.

Case 1: Start of invocation. If the action is *invoc*, we choose the single-invocation trace to consist only of this action. Below we reproduce the rules for *invoc* in the interleaving and in the single-invocation semantics.
invocation:

$localState\ S\ i = None \wedge$
$procedure\ (prog\ S)\ proc = (initialState,\ impl) \wedge$
$uniqueIds\ proc \subseteq knownIds\ S \wedge invocOp\ S\ i = None \Longrightarrow$
$S \xrightarrow{(i,\ AInvoc\ proc)} S(\!|localState := localState\ S(i \mapsto initialState),$
$currentProc := currentProc\ S(i \mapsto impl),$
$visibleCalls := visibleCalls\ S(i \mapsto \varnothing),$
$invocOp := invocOp\ S(i \mapsto proc)|\!)$

s-invocation:

$invocOp\ S\ i = None \wedge$
$procedure\ (prog\ S)\ proc = (initState,\ impl) \wedge$
$uniqueIds\ proc \subseteq knownIds\ S' \wedge$
$state\text{-}wellFormed\ S' \wedge$
$(\forall\ tx.\ txStatus\ S'\ tx \neq Some\ Uncommitted) \wedge$
$invariant\text{-}all\ S' \wedge$
$invocOp\ S'\ i = None \wedge$
$prog\ S' = prog\ S \wedge$
$S'' = S'$
$(\!|localState := localState\ S'(i \mapsto initState),$
$currentProc := currentProc\ S'(i \mapsto impl),$
$visibleCalls := visibleCalls\ S'(i \mapsto \varnothing),$
$invocOp := invocOp\ S'(i \mapsto proc)|\!) \wedge$
$valid = invariant\text{-}all\ S'' \wedge (\forall\ tx.\ txOrigin\ S''\ tx \neq Some\ i) \Longrightarrow$
$S \xrightarrow{(i,\ AInvoc\ proc,\ valid)} S''$

If we choose S' in the single-invocation semantics to be the pre-state in interleaving semantics, we get that the resulting states are equal. It remains to be shown that the additional assumptions in the *S-invocation* rule compared to the *invocation* rule are fulfilled.

Well-formedness of S' is given as it is reachable in the interleaving semantics. Likewise, the invariant holds in S' as we assumed this initially.

We can also show that there are no uncommitted transactions: As our trace is packed and does not contain any context switches in transactions, the only active transaction can be in the invocation of the last action in the trace. However, we are considering an *invoc* in the next step, so there cannot yet be an active transaction on invocation i. Thus, there can be no active transaction. The other preconditions of rule *S-invocation* can be shown from the fact that it also is the start of an invocation in the interleaving semantics.

Case 2: Start of transaction. From the coupling invariant we know that the *state-monotonicGrowth* predicate holds between the state of the single-invocation semantics and the state of the interleaving semantics and with respect to invocation i. This allows us to use the *beginAtomic* rule of the single-invocation semantics (see Figure 5.8 for a comparison of the two rules), instantiating S' in the rule with the state of the interleaving semantics before starting the transaction. The state S'' after starting the transaction is then again equivalent for both executions.

We then need to show that the remaining prerequisites of the *S-atomic* rule are fulfilled.

a) There are no uncommitted transactions in S'. Since the trace is packed and does not contain any context switches, by Lemma 5.3.6.3 there can be at most one current transaction in state S''. As this is the newly started transaction, there cannot be any uncommitted transaction in state S'.

b) The states S' and S'' are *wellformed*. This follows from the fact that they are part of the execution in the interleaving semantics.

c) The transaction snapshot is a *consistentSnapshot*. We can first prove that the snapshot is consistent in S'' using Lemma 5.3.1.3. From this it follows that the snapshot is also consistent in S'.

d) There are not yet any calls in the new transaction. This follows from the fact that the *txStatus* is undefined in state S' and Lemma 5.3.1.5.

Case 3: Invariant check. In the case of an invariant check in the interleaving semantics, there is no state-change and therefore we simply reuse the single-invocation trace from the induction hypothesis and are done.

Other cases. The action is not a valid context switch – start of invocations and transactions are already handled above. As the trace is *packed* by assumption, we therefore know that the previous action in the trace was also on

beginAtomic:

$$
\begin{array}{l}
\mathit{localState}\ S\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S\ i \triangleq f \wedge \\
f\ \mathit{ls} = \mathit{BeginAtomic}\ \mathit{ls}' \wedge \\
\mathit{currentTx}\ S\ i = \mathit{None} \wedge \\
\mathit{txStatus}\ S\ t = \mathit{None} \wedge \\
\mathit{visibleCalls}\ S\ i \triangleq \mathit{vis} \wedge \mathit{chooseSnapshot}\ \mathit{snapshot}\ \mathit{vis}\ S \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{ABeginAtomic}\ t\ \mathit{snapshot})} S(\lfloor \mathit{localState} := \mathit{localState}\ S(i \mapsto \mathit{ls}'), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{currentTx} := \mathit{currentTx}\ S(i \mapsto t), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{txStatus} := \mathit{txStatus}\ S(t \mapsto \\
\mathit{Uncommitted}), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{txOrigin} := \mathit{txOrigin}\ S(t \mapsto i), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{visibleCalls} := \mathit{visibleCalls}\ S(i \mapsto \\
\mathit{snapshot})\rfloor)
\end{array}
$$

s-beginAtomic:

$$
\begin{array}{l}
\mathit{localState}\ S\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S\ i \triangleq f \wedge \\
f\ \mathit{ls} = \mathit{BeginAtomic}\ \mathit{ls}' \wedge \\
\mathit{currentTx}\ S\ i = \mathit{None} \wedge \\
\mathit{txStatus}\ S\ t = \mathit{None} \wedge \\
\mathit{prog}\ S' = \mathit{prog}\ S \wedge \\
\mathit{state\text{-}monotonicGrowth}\ i\ S\ S' \wedge \\
\mathit{invariant\text{-}all}\ S' \wedge \\
(\forall\ \mathit{tx}.\ \mathit{txStatus}\ S'\ \mathit{tx} \neq \mathit{Some}\ \mathit{Uncommitted}) \wedge \\
\mathit{state\text{-}wellFormed}\ S' \wedge \\
\mathit{state\text{-}wellFormed}\ S'' \wedge \\
\mathit{localState}\ S'\ i \triangleq \mathit{ls} \wedge \\
\mathit{currentProc}\ S'\ i \triangleq f \wedge \\
\mathit{currentTx}\ S'\ i = \mathit{None} \wedge \\
\mathit{visibleCalls}\ S\ i \triangleq \mathit{vis} \wedge \\
\mathit{visibleCalls}\ S'\ i \triangleq \mathit{vis} \wedge \\
\mathit{chooseSnapshot}\ \mathit{vis}'\ \mathit{vis}\ S' \wedge \\
\mathit{consistentSnapshot}\ S'\ \mathit{vis}' \wedge \\
\mathit{txStatus}\ S'\ t = \mathit{None} \wedge \\
(\forall\ c.\ \mathit{callOrigin}\ S'\ c \neq \mathit{Some}\ t) \wedge \\
\mathit{txOrigin}\ S'\ t = \mathit{None} \wedge \\
S'' = S' \\
(\lfloor \mathit{txStatus} := \mathit{txStatus}\ S'(t \mapsto \mathit{Uncommitted}),\ \mathit{txOrigin} := \mathit{txOrigin}\ S'(t \mapsto i), \\
\quad \mathit{currentTx} := \mathit{currentTx}\ S'(i \mapsto t),\ \mathit{localState} := \mathit{localState}\ S'(i \mapsto \mathit{ls}'), \\
\quad \mathit{visibleCalls} := \mathit{visibleCalls}\ S'(i \mapsto \mathit{vis}')\rfloor) \Longrightarrow \\
S \xrightarrow{(i,\ \mathit{ABeginAtomic}\ t\ \mathit{vis}',\ \mathit{True})} S''
\end{array}
$$

Figure 5.8.: *Comparison of the rules for starting a transaction in the interleaving semantics (top) and single-invocation semantics (bottom).*

the same invocation i. Using the coupling invariant we thus know that the interleaving semantics and single-invocation semantics are both in the same state S' before executing the current action.

The interleaving semantics now takes a step to state S'' using action a. We then show that the single-invocation semantics also goes to the same state S'' using the same action a. Since in the remaining cases, the rules for the single-invocation semantics are similar to the rules of the interleaving semantics, the proof is straightforward. □

We have now shown, that we can convert a trace with no invariant violations to the single-invocation semantics. Next we show that we can convert a single step which makes the invariant fail to an equivalent step in the single-invocation semantics.

Lemma 5.3.7.2 (Convert step with failing invariant) Assume we have a step in the interleaving semantics where the pre-state satisfies the invariant but the post-state does not. Then we can perform the same step in the single-invocation semantics starting from a coupled state, leading to the same state, and also producing an invariant-violation.

Moreover, we assume the same properties of the trace that we assumed in Lemma 5.3.7.1, albeit reduced to a single action.

lemma *convert-to-single-session-trace-invFail-step*:
fixes $tr :: ('proc::valueType, 'op, 'any::valueType)\ trace$
and $i :: invocId$
and $S\ S' :: ('proc, 'ls, 'op, 'any)\ state$
assumes *step*: $S \xrightarrow{(i,a)} S'$
and *S-wellformed*: *state-wellFormed S*
and *noFails*: $a \neq ACrash$
— invariant holds in the initial state
and *inv*: *invariant-all S*
— invariant no longer holds
and *not-inv*: $\neg$*invariant-all* S'
and *coupling*: *state-coupling S S2 i sameSession*
and *ctxtSwitchCases*: $\neg$*sameSession* $\Longrightarrow$ *allowed-context-switch a*
and *noUncommitted*: $\bigwedge p.\ a = AInvoc\ p \Longrightarrow \forall tx.\ txStatus\ S\ tx \neq Some\ Uncommitted$
shows $(S2 \xrightarrow{(i,\ a,\ False)} S')$

We prove the lemma by case distinction over action a. For most actions the value of the invariant cannot change, since invariants may only refer to global state from committed transactions. Therefore, we only need to consider the cases for *invoc*, *return*, and *endAtomic* in detail.

Case *invoc*: Because of the coupling invariant we get that *localState* and *invocationOp* are undefined for invocation i in state $S2$, like they are in S. With Lemma 5.3.1.4 we also infer that there are not yet any transactions in state S. With these prerequisites, we can show that the step in the single-invocation semantics is possible.

Case *return*: Since *return* is not an allowed context switch, *sameSession* must be true. From the coupling invariant we thus get that $S2 = S$. Then, it is straightforward to show that the step can be done in the single-invocation semantics.

Case *endAtomic*: Similar to the case above, *endAtomic* is not an allowed context switch, and thus $S2 = S$. Again, the corresponding step in the single-invocation semantics can then be shown automatically. □

The next Lemma is similar to the Lemma we just proved, but works with a trace instead of a single action.

Lemma 5.3.7.3 (Convert trace with failing invariant)

lemma *convert-to-single-session-trace-invFail*:
fixes *tr* :: ('*proc*::*valueType*, '*op*, '*any*::*valueType*) *trace*
and *S S'* :: ('*proc*, '*ls*, '*op*, '*any*) *state*
assumes *steps*: $S \xrightarrow{tr}* S'$
and *S-wellformed*: *state-wellFormed S*
and *packed*: *packed-trace tr*
and *noFails*: $\bigwedge s.\ (s, ACrash) \notin set\ tr$
and *noUncommittedTx*: $\bigwedge tx.\ txStatus\ S\ tx \neq Some\ Uncommitted$
and *noContextSwitches*: $\neg contextSwitchesInTransaction\ tr$
— invariant holds in the initial state
and *inv*: *invariant-all S*
— invariant no longer holds
and *not-inv*: $\neg$ *invariant-all S'*
and *noAssertionErrors*: $\bigwedge a.\ a \in set\ tr \Longrightarrow get\text{-}action\ a \neq ALocal\ False$
shows $\exists tr'\ S2\ s.\ (S \xrightarrow{(s,\ tr')}* S2)$
$\wedge\ (\exists a.\ (a, False) \in set\ tr')$

For the proof, let tr_1 be the longest prefix of the trace tr not containing an invariant violation, S_1 be the state after executing tr_1, a be the action in tr after tr_1 and S_{fail} the state after executing a. This means that a is the action making the invariant fail and S_{fail} is the first state breaking the invariant.

We can now use Lemma 5.3.7.1 to convert tr_1 to a single-invocation execution with trace tr_1' and final state S_1'. We use the invocation of action a as the invocation of the execution. According to the Lemma, the state S_1' is in the coupling relation with state S_1.

Next we can use Lemma 5.3.7.2 to convert the step $S_1 \xrightarrow{a} S_{fail}$ to a step in the single-invocation semantics $S_1' \xrightarrow{a,False}\cdot S_{fail}$. To apply the Lemma, we have to show that there are no uncommitted transactions in the case that a is the start of an invocation.

To show this we again use the fact that we have a packed trace without context switches in transactions. Therefore, the only active transaction in state S_{fail} could be on the invocation that just started with action a. However, the initial state of an invocation cannot contain any active transactions. □

We can now use this Lemma to show that it is sufficient to consider the single-invocation semantics for verifying programs.

Lemma 5.3.7.4 (Show correctness via single-invocation semantics) If a program is correct with respect to the single-invocation semantics and the invariant holds in the initial state of the program, then the program is also correct with respect to the interleaving semantics.

theorem *show-correctness-via-single-session*:
assumes *works-in-single-session*: *programCorrect-s program*
and *inv-init*: *invariant-all* (*initialState program*)
shows *programCorrect program*

We first use Lemma 5.3.6.2 to reduce the verification problem to a single trace tr that is *packed*, does not contain crashes, and where all transactions end. We show that this given trace is correct using a proof by contradiction.

Assume tr contains an invariant violation and let S_{fail} be the first state in the execution of tr violating the invariant. With this prefix of tr, we can use Lemma 5.3.7.3 to obtain a failing trace in the single-invocation semantics. However, we assumed that all single-invocation executions are correct so we have a contradiction. □

5.4. Completeness

We have now derived a proof technique for highly available applications and proven its soundness. This begs to ask the question: Is the technique complete as well?

We now show that the reduction from the interleaving semantics to the single-invocation semantics preserves correctness. If a program is correct with respect to the interleaving semantics (*programCorrect*), then it is also correct with respect to the single-invocation semantics (*programCorrect-s*).

theorem *completeness*:
assumes *correct*: *programCorrect program*
shows *programCorrect-s program*

Thus, if we can prove correctness with respect to the interleaving semantics in Isabelle, we can always use this Lemma to obtain an Isabelle proof for the correctness with respect to the single-invocation semantics.

We prove this theorem by contraposition. First, assume the program is not correct with respect to the single-invocation semantics. Then we can find a single-invocation execution with an incorrect trace. Let a be an action in the trace containing an error and let S be the state before action a.

We can then show that the state S is *wellformed* because every state reachable in the single-invocation semantics is also reachable in the interleaving semantics. This is a result of how we designed the single-invocation semantics. Every rule that could otherwise violate this property (which are the rule for starting a procedure invocation and for starting a transaction) has an additional assumption that the state is *wellformed*.

We now have the situation that executing the step $S \xrightarrow{i,a,false} S'$ in the single-invocation semantics either causes an assertion failure ($a = local(False)$)

or the state S' does not satisfy the program invariant. We then have to show that we can also produce an error in the interleaving semantics.

As errors in the single-invocation semantics can only occur when the action is *invoc*, *endAtomic*, *local*, or *return*, we only have to consider these four cases. In the case of a procedure invocation (*invoc*), we have the assumption that there is a well-formed state S_i, from which the new state S' is constructed. We can show that the step from S_i to S' corresponds to an invocation step in the interleaving semantics. Thereby we can reproduce the error. For the other cases, the situation is similar: In these cases, it even is possible to reproduce the exact same step and get $S \xrightarrow{i,a} S'$ in the interleaving semantics.

□

With this we have shown that the essential step of our technique maintains completeness of the technique. However, one needs to consider that an essential step in the completeness proof is that every state reachable in the single-invocation semantics is well-formed, i.e. reachable in the interleaving semantics. We achieved this by including it as an explicit premise in the rules for *invoc* and *beginAtomic*. Yet in the tool implementation (see Chapter 7) we cannot include this semantic property and have to over-approximate it with first-order formulas. Thereby we loose some completeness depending on how precisely we can capture the reachability property. As this is hard to quantify formally, we have not researched this in more detail.

Chapter 6

Proof Automation in Isabelle

In the previous chapter we have formalized the system semantics and the core of our proof technique. As a step towards tool supported verification, in this chapter we develop proof rules suitable for symbolic execution. To this end we first define a concrete programming language embedded in Isabelle in Section 6.1. Then we define the proof rules for symbolic execution in Section 6.2. Finally, we discuss the handling of unique identifiers in Section 6.3 and end the chapter with a discussion about the completeness aspects of the proof rules in Section 6.4.

6.1. A Shallow Embedding of a Programming Language

In Chapter 5 we considered a semantics that leaves the actual implementation language for procedures abstract. We merely modeled procedure implementations as arbitrary, deterministic state machines, with a function that takes the current local state and produces an action and, where required, a continuation to take the results of the actions and produce a new local state.

In order to apply our technique to concrete examples, we now concretize the state machine by defining a language that can be translated to the abstract state machine framework. To define the language, we use a shallow encoding using Isabelle's Monad syntax.

Using concrete language constructs allows us to then define proof rules for each language construct. Our proof rules are structured such that they can be used to simulate symbolic execution [Sch16]. This allows us to derive verification conditions from a given procedure implementation and the program invariants.

The programming language we use for the embedding is a minimal imperative language. We give an informal abstract syntax for the language in Figure 6.1. The language contains two basic control flow constructs: An if statement and a universal looping construct. To support mutable variables, we include a construct to create mutable references, which can then be updated in assignments ($:\leftarrow$) and read with the respective constructs (*read*). Finally, we include

two constructs for interacting with databases: Atomic blocks group the enclosed database operations into an atomic transaction and the *call* statement is used to perform calls to the database.

$$
\begin{aligned}
\mathit{Stmt} \rightarrow\ & \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{Stmt}\ \textbf{else}\ \mathit{Stmt} \\
\mid\ & \textbf{while}\ \mathit{Expr}\ \mathit{Stmt} \\
\mid\ & \textbf{return}\ \mathit{Expr} \\
\mid\ & \mathit{Stmt};\ \mathit{Stmt} \\
\mid\ & \mathit{Var} \leftarrow \textbf{makeRef}\ \mathit{Expr} \\
\mid\ & \mathit{Expr}\ :\!\leftarrow\ \mathit{Expr} \\
\mid\ & \mathit{Var} \leftarrow \textbf{read}\ \mathit{Expr} \\
\mid\ & \textbf{atomic}\ \mathit{Stmt} \\
\mid\ & \mathit{Var} \leftarrow \textbf{call}\ \mathit{Expr} \\
\mid\ & \mathit{Var} \leftarrow \textbf{new}\ \mathit{Expr}
\end{aligned}
$$

Figure 6.1.: *Informal abstract syntax of the embedded programming language.*

6.1.1. Shallow vs Deep Embeddings

There are two styles for embedding a language within Isabelle/HOL: deep and shallow embedding [WN04]. In a deep embedding, the language is modelled using datatypes and the semantics of the language is defined separately. In the shallow embedding style, the language constructs are directly defined using constructs of the host language.

Shallow embeddings typically have more concise definitions as the intermediate syntactical level is skipped. For users, it also has the benefit that all features of the host language can be used.

However, shallow embeddings do not provide a programmatic representation of the program structure. Therefore, it is not possible to write functions that take a syntax tree as input – all definitions need to work on the semantic level. Deep embeddings also have the benefit that they can be independent of constraints in the host language, whereas shallow embeddings need to follow the usual typing rules.

For modeling our language, we chose a shallow embedding, mostly in order to keep the definitions more concise. The lack of a syntactic representation of programs is negligible for our use case: We are interested in the structure when applying proof rules, but those can still work on the structure of Isabelle terms and thereby on the program structure. Using Isabelle's method definition language Eisbach [MMW16], we can even write methods operating on the structure of terms by combining other methods.

To keep the representation of programs close to the imperative program text as shown in Section 2.1, we use Isabelle's monadic syntax [Bul+08]. For

```
definition getMessage-impl :: val ⇒ (val,operation,val) io where
  getMessage-impl m ≡  do {
  atomic (do {
    exists ← call (Message (KeyExists m));
    if exists = Bool True then do {
      author ← call (Message (NestedOp m (Author Read)));
      content ← call (Message (NestedOp m (Content Read)));
      return (Found author content)
    } else do {
      return NotFound
    }
  })
}
```

Figure 6.2.: *Implementation of getMessage in Isabelle using monad syntax.*

example, the function *getMessage* from our case study can be modelled in Isabelle as shown in Figure 6.2. While this choice allows us to reuse existing syntactical definitions from Isabelle, it gives rise to some challenges regarding Isabelle's limited type system. In the following, we show how we can work around these limitations while still guaranteeing that programs are type safe.

6.1.2. Language Definition as a Monad

A monad in functional programming is a type m that takes one type parameter and supports two operations:

- The *bind* operation has type $\alpha\ m \Rightarrow (\alpha \Rightarrow \beta\ m) \Rightarrow \beta\ m$ and is denoted by the infix operator $\gg=$.
- The *return* operation has type $\alpha \Rightarrow \alpha\ m$.

These operations must satisfy the three monad laws [Wad90]:

Left identity : $(return\ a \gg= f) \equiv f\ a$

Right identity : $(m \gg= return) \equiv m$

Associativity : $((m \gg= f) \gg= g) \equiv (m \gg= (\lambda x.\ f\ x \gg= g))$

The equivalence operator $\equiv$ above is defined as observational equivalence, i.e. $A \equiv B$ if A and B can be replaced with each other in any program context without changing the behavior of the program.

One typical example for monads are container types. In this case, *return* x produces a container holding a single value x and $m \gg= f$ applies f on all values in m and combines the resulting containers to a new container. For example lists form a monad with $return\ x = [x]$ and $([x_1, \ldots, x_n] \gg= f) = f\ x_1 \cdot \ldots \cdot f\ x_n$.

Another typical example for monads are actions like the *IO* monad in Haskell. Here, *return* x transforms a pure value into an action computing

value x and the bind operation composes two actions sequentially, using the result of the first action to determine the second action.

The support for monads in Isabelle/HOL is limited. As there is no higher kinded polymorphism, it is not possible to define a *Monad* typeclass as Haskell does and it is not possible to write generic functions on monads. However, the concept is still part of the standard library and supports a common notation for monads. The bind operator ($\gg=$) can be set up to use ad-hoc overloading which allows it to be used with different types of monads. Moreover, there is a Haskell-like do notation [Pey03]. This allows to use the bind-notation in a way that is more similar to imperative programming languages. In Figure 6.3 we summarize the notation and translations used by Isabelle's monad syntax.

Syntax:

$$
\begin{aligned}
&S \to \textbf{do}\ \{B\}\\
&B \to \mathit{Pattern} \leftarrow \mathit{Expression}\\
&B \to \textbf{let}\ \mathit{Pattern} = \mathit{Expression}\\
&B \to \mathit{Expr}\\
&B \to B;\ B
\end{aligned}
$$

Transformations:

$$
\begin{aligned}
\textbf{do}\ \{e_1;\ e_2\} &\rightharpoonup e_1 \gg= (\lambda\_.\ e_2)\\
\textbf{do}\ \{p \leftarrow e_1;\ e_2\} &\rightharpoonup e_1 \gg= (\lambda p.\ e_2)\\
\textbf{do}\ \{\textbf{let}\ p = e_1;\ bs\} &\rightharpoonup \textbf{let}\ p = e_1\ \textbf{in}\ \textbf{do}\ \{bs\}\\
\textbf{do}\ \{b;\ c;\ cs\} &\rightharpoonup \textbf{do}\ \{b;\ \textbf{do}\ \{c;\ cs\}\}\\
\textbf{let}\ p = e_1;\ e_2 &\rightharpoonup \textbf{let}\ p = e_1\ \textbf{in}\ e_2\\
\textbf{do}\ \{e\} &\rightharpoonup e
\end{aligned}
$$

Figure 6.3.: *Isabelle's Monad Syntax (see Monad_Syntax.thy in the Isabelle distribution [NPW02]).*

For embedding our language into Isabelle, we define a monadic type *io*. Unfortunately, Isabelle imposes some restrictions on the definition of datatypes which slightly obscures our formalization. For this reason we first present a simplified version of the datatype in Haskell to make the concepts and our design decisions clear. The corresponding type is shown in Figure 6.4. A value of type `Io` describes actions working on a procedure-local data store of type `Store`, which represents the state of mutable local variables. We will later extend the datatype to also include actions for interacting with the distributed database, however that is an orthogonal aspect and can thus be omitted for the discussion here. The simplified datatype definition consists of four cases:

LocalStep This case handles interactions with the local store. It includes a function that takes the complete store as input and computes a new

store and a result value. Higher level operations like updating a single variable can then be defined on top of this construct.

Loop There are two approaches for modelling loops within a monad. One approach is to use the recursion from the host language to construct infinite monadic values. This is the idea behind the looping constructs offered by Haskell's `Control.Monad.Loops` package [Coo15]. However, this approach would not be transferable to Isabelle, where we cannot work with recursive functions without a proof of well-definedness (e.g. termination). We therefore chose the second approach, which is to equip the monadic type with a construct for loops.

As in the previous case, we want to use the loop statement as a basis and later define higher level constructs on top. Therefore, the loop construct must be general enough to allow all other typical loop constructs to be translated to it. While `while`-loops are worthwhile to be considered as universal looping constructs, it would require additional mutable variables to translate something like a `foreach`-loop to a `while`-loop. We want to avoid this, since it would affect invariants expressed over the local variable store. Instead, we allow a value to be passed between loop iterations. If the loop body produces a value `Continue x`, then the loop is repeated with value `x`. To end iterating, the loop body must produce a value `Break x`, in which case the value `x` is also the result of the overall loop construct. This is similar to the `loop` function from Haskell's extra package [Mit20].

To show how typical loop constructs can be translated, Figure 6.4 includes examples for a `while`- and a `foreach`-loop in line 35.

Seq This constructs enables sequential composition of two actions and can thus be directly used as the implementation of the $\gg=$operation. When executed, the result of the first action is used to determine the second action.

Return This case allows us to transform pure values into the type `Io`.

Finally, we want to execute a program stepwise, when it is given as an `Io`-value. This way, we can interleave executions of several concurrent procedure invocations, and we can later add interaction with the external world (e.g. the database) between steps. To show that stepwise execution is possible with our datatype definition, we give a function `ioStep` in line 21 of Figure 6.4.

As mentioned earlier, this Haskell definition of the `Io` type cannot be directly transferred to Isabelle. Even in Haskell, it requires to enable the generalized algebraic datatypes (GADT) extension to work. There are two places where simple datatypes are not sufficient:

1. The `Seq` case contains values of type `Io a` in the first action and `Io b` in the second.

```haskell
data Io a where
  LocalStep :: (Store -> (Store, a)) -> Io a
  Loop :: b -> (b -> Io (LoopResult b a)) -> Io a
  Seq :: Io b -> (b -> Io a) -> Io a
  Return :: a -> Io a

instance Functor Io where
  fmap f x = Seq x (\x -> Return (f x))

instance Applicative Io where
  pure = Return
  liftA2 f x y = Seq x (\xr -> Seq y (\yr -> Return (f xr yr)))

instance Monad Io where
  (>>=) = Seq

data LoopResult a b = Continue a | Break b

type Store = Map Int Dynamic

ioStep :: Store -> Io a -> (Store, Either (Io a) a)
ioStep store (LocalStep f) =
  let (store', res) = f store in
  (store', Right res)
ioStep store (Loop i bdy) =
  (store, Left (Seq (bdy i) cont))
  where cont (Continue x) = Loop x bdy
        cont (Break x) = Return x
ioStep store (Seq a b) =
  case ioStep store a of
  (store', Left a') -> (store', Left (Seq a' b))
  (store', Right a) -> (store', Left (b a))
ioStep store (Return x) = (store, Right x)

-- Syntactic sugar for loops:
while :: Io Bool -> Io () -> Io ()
while condition body = Loop () $ \_ -> do
  c <- condition
  if c then do
    body
    return (Continue ())
  else
    return (Break ())

foreach :: [a] -> (a -> Io ()) -> Io ()
foreach list body = Loop list $ \list -> do
  case list of
   [] -> return (Break ())
   (x:xs) -> do
      body x
      return (Continue xs)
```

Figure 6.4.: *Definition of our `Io` type in Haskell.*

2. The `Loop` case contains a value of type `Io (LoopResult a b)` in the loop body.

We need to rewrite these two cases to express the `Io` type in Isabelle. One way to avoid limitations in the type system is to fall back to dynamic types. Haskell includes a type `Dynamic` with functions `toDyn :: Typeable a => a -> Dynamic` and `fromDynamic` :: Typeable a => Dynamic -> Maybe a!, which can be used to do this. The `Typeable a` constraint requires that a runtime representation can be generated for the type `a`, which can be used for dynamic type checking in `fromDynamic`. Using the `Dynamic` type, we can rewrite the case for loops from `Loop :: b -> (b -> Io (LoopResult b a))-> Io a` to `Loop :: Dynamic -> Dynamic -> Io a`. Here, we have replaced the type parameter `b` with the type `Dynamic`. Also, we replaced the complete loop body with a value of `Dynamic` that represents a function `(Dynamic -> Io (LoopResult Dynamic a))`.

While it is not possible to have a type like `Dynamic` in Isabelle[1], there are some less powerful alternatives: The typeclass `countable` includes all types with an injective function into the natural numbers. This means that it includes all types where all values have a finite representation. However, it does not contain function types where the domain is infinite and the range contains more than one element. As our datatype `Io` contains such functions, it is not itself countable and therefore we cannot encode the loop body using a single natural number.

An alternative is offered by Paulson's formalization of Zermelo Fraenkel Set Theory Isabelle/HOL [Pau19]. This development adds a new type `V`, which is big enough to allow injective mappings from basically any HOL type into `V`. As such a type cannot be defined using conservative extensions, the theory uses axioms to describe the type. The axioms follow the axioms of Zermelo Fraenkel Set Theory, which should ensure that no inconsistencies arise from the additional axioms[2].

For our purposes, the relevant restrictions are given using the two typeclasses `small` and `embeddable`. Intuitively, the class `small` contains all HOL types that do not contain the type `V` itself. In contrast, the class `embeddable` may contain the type `V`. In particular, `V` itself is `embeddable` and functions with a `small` domain and an `embeddable` range are `embeddable`. Of course, finite combinations like product and sum types of `embeddable` types are also `embeddable`.

For our type `Io`, this means that we can make the type `Io` a member of the type class `embeddable` and thus encode the loop body as a value of type `V`. However, we cannot use type `V` for the input value of the loop body as that would put V into the range of a function we want to embed into `V`. We therefore

[1] The function `toDyn` can be instantiated to type `(Dynamic -> Bool)-> Dynamic` and it must be injective as the reverse function `fromDynamic` exists. However the existence of such a function would immediately lead to an inconsistency using Cantor's theorem (theorem `Cantors_paradox` in the Isabelle standard library): There can be no surjective function from a set to its powerset and vice versa there can be no injective function from the powerset to the set itself. Haskell avoids this inconsistency because all values have a finite representation (their value in memory), so the type `a -> Bool` is not actually the powerset of `a`.

[2] However, I am not aware of a proof for this when combined with the HOL axioms.

resort to the typeclass `countable` to keep the type of this value flexible. Since this value is typically only used for program values, which tend to have a finite representation, this is not a major restriction.

We could use a similar solution for the `Seq` case. However, this would restrict the bind operation and only allow values of a countable type to be passed from the first action to the second. Instead, we use a continuation passing style for encoding sequential composition. We do this by adding a continuation of type `Io` to each case of the datatype. For example the type for `LocalStep` is changed from "`(Store -> (Store, a))-> Io a`" to "`(Store -> (Store, Io a))-> Io a`". With this change, there are no longer any additional type parameters in the cases of the datatype and every recursive occurrence of `Io` has an unchanged type parameter `a`.

This allows us to express the datatype in Isabelle. The resulting definition is shown in Figure 6.5. The Isabelle type is parameterized by the return type ('a) of the action, the type for database operations ('operation), and the type of program values ('any).

Besides the features of the simplified `Io` type introduced above, this type also contains actions for interacting with the database and for generating unique identifiers. The actions correspond to the actions that can be taken by the abstract state machine as defined in Section 5.1. Only the Loop construct is added since arbitrary loops cannot be expressed otherwise with a finite datatype[3]. We briefly recap the purpose of each case:

WaitLocalStep This action performs an operation, which can read and update the local variable store. The computation is expressed with a function that takes the current store as input and returns a boolean, the new state of the store, and an action to continue with. The boolean is used to model local assertions and is false if there is an assertion violation in the local action.

WaitBeginAtomic This action starts an atomic transaction.

WaitEndAtomic This action commits an atomic transaction.

WaitDbOperation This action performs an operation on the database. The result of the database operation is passed into a continuation that determines the next action.

WaitNewId This action generates a fresh, globally unique identifier. The first parameter is a predicate, which restricts the unique identifier to be generated. This can be used to generate only identifiers of a certain type. The second parameter is a continuation that takes the generated identifier and uses it to compute the next action.

WaitReturn This case is for returning a value.

[3] It might be possible with a co-datatype [Bla+14], but this would complicate the definition of the bind operation and related proofs.

Loop This action implements loops with a loop state of type `'any`. The first parameter defines the initial loop state. The second parameter is defined using the type `V`, but actually represents a value of type `'any => ('any LoopResult, 'operation, 'any)io`. It takes the current loop state and returns the action for the loop body. If this action returns `Continue x`, the loop is repeated with `x` as the new loop state. Otherwise, if the action returns `Break x`, the loop is finished and the value `x` is passed to the continuation in the last parameter of the `Loop` case to determine the next action.

```
type-synonym iref = nat
type-synonym 'v store = iref ⇀ 'v

datatype ('a,'op, 'any) io =
    WaitLocalStep 'any store ⇒ bool × 'any store × ('a,'op, 'any) io
  | WaitBeginAtomic ('a,'op, 'any) io
  | WaitEndAtomic ('a,'op, 'any) io
  | WaitNewId 'any ⇒ bool 'any ⇒ ('a,'op, 'any) io
  | WaitDbOperation 'op 'any ⇒ ('a,'op, 'any) io
  | WaitReturn 'a
  | Loop 'any V 'any ⇒ ('a,'op, 'any) io

function bind (infixl ≫=io 54) where
  WaitLocalStep n ≫=io f = (WaitLocalStep (λs. let (a,b,c) = n s
                                           in (a, b, bind c f)))
| WaitBeginAtomic n ≫=io f = (WaitBeginAtomic (n ≫=io f))
| WaitEndAtomic n ≫=io f = (WaitEndAtomic (n ≫=io f))
| WaitNewId P n ≫=io f = (WaitNewId P (λi. n i ≫=io f))
| WaitDbOperation op n ≫=io f = (WaitDbOperation op (λi. n i ≫=io f))
| WaitReturn s ≫=io f = (f s)
| Loop i body n ≫=io f = (Loop i body (λx. n x ≫=io f))
```

Figure 6.5.: *Definition of the monadic type io and its bind operation.*

As we no longer have a `Seq` case as in the simplified version of `Io`, the bind (≫=) operation must do more work than previously: It is now implemented recursively, by binding the right action into the continuation of the first action. This terminates as datatypes are finite in Isabelle and therefore we will eventually reach the *return* case. The corresponding implementation is shown in Figure 6.5.

We can then show that our definition of *bind* and *return* (defined here as *WaitReturn*) satisfy the monad laws introduced at the beginning of the section (page 87). The proof is done by induction over the leftmost variable.

We also define these laws as automatic simplifications. This ensures that programs are always in a normalized form, either as a single action a or an atomic action followed by a remaining program ($a \gg\!= S$). This makes it easier to define proof rules on the program structure in the next Section 6.2.

Using the definition of *io* monad above, we can now define higher level

language constructs as monadic values by using simple definitions. Figures 6.6 and 6.7 show the basic language constructs of the Repliss language. For the continuation in the *io* datatype, we use a simple *WaitReturn* action that returns the result of the action (or unit if there is no result). Note that this continuation is later changed when combining several actions via *bind*. As the *while*-loop does not require a loop variable, its definition uses the expression "*???*" in its place, which stands for an undefined value.

For interacting with the local variable store, we use typed references. A typed reference can be created for every *countable* type. The *assign* action updates a reference and the *read* action reads the current value of a reference.

```
beginAtomic ≡ WaitBeginAtomic (WaitReturn ())

endAtomic ≡ WaitEndAtomic (WaitReturn ())

newId P ≡ WaitNewId P WaitReturn

call op ≡ WaitDbOperation op WaitReturn

return x ≡ WaitReturn x

skip ≡ return default

atomic f ≡ beginAtomic ≫= (λ-. f ≫= (λr. endAtomic ≫= (λ-. return r)))

makeRef v ≡
WaitLocalStep
 (λs. let r = freshRef (dom s)
      in (True, s(r ↦ intoAny v), WaitReturn (Ref r)))

read ref ≡
WaitLocalStep
 (λs. case s (iref ref) of None ⇒ (False, s, WaitReturn (from-nat 0))
      | Some v ⇒ (True, s, WaitReturn (fromAny v)))

ref :← v ≡
WaitLocalStep
 (λs. case s (iref ref) of None ⇒ (False, s, WaitReturn ())
      | Some x ⇒ (True, s(iref ref ↦ intoAny v), WaitReturn ()))

update ref upd ≡
WaitLocalStep
 (λs. case s (iref ref) of None ⇒ (False, s, WaitReturn ())
      | Some v ⇒
          (True, s(iref ref ↦ intoAny (upd (fromAny v))), WaitReturn ()))
```

Figure 6.6.: *Definition of language constructs using the io monad.*

To see how the language constructs work together in the *io* monad, consider again the example from Figure 6.2. If we desugar the do-notation, then

```
loop init body ≡
Loop (intoAny init)
 (loop-body-to-V
   (λacc. body (fromAny acc) ≫=
          (λres. return
                   (case res of Continue x ⇒ Continue (intoAny x)
                    | Break x ⇒ Break (intoAny x)))))
 (return ∘ fromAny)

while body ≡
Loop ???
 (loop-body-to-V
   (λ-. body ≫= (λx. return ((if x then Break else Continue) ???))))
 (return ∘ (λ-. ()))

forEach elements body ≡
loop (elements, [])
 (λ(elems, acc).
    case elems of [] ⇒ return (Break (rev acc))
    | x · xs ⇒ body x ≫= (λr. return (Continue (xs, r · acc))))
```

Figure 6.7.: *Definition of language constructs for loops using the io monad.*

getMessage-impl m simplifies to the term in Figure 6.8.

```
getMessage-impl m ≡
atomic
 (call (Message (KeyExists m)) ≫=
  (λexists.
     if exists = Bool True
     then call (Message (NestedOp m (Author Read))) ≫=
          (λauthor.
             call (Message (NestedOp m (Content Read))) ≫=
             (λcontent. return (Found author content)))
     else return NotFound))
```

Figure 6.8.: *Desugared version of getMessage from Figure 6.2.*

To make the above definitions usable in our generic framework, we need to convert a value of type *io* to a state machine. Remember that we defined a procedure implementation as a function that takes a local state *'ls* and returns an action to perform:

```
type-synonym ('ls, 'op, 'any) procedureImpl =
  'ls ⇒ ('ls, 'op, 'any) localAction
```

We now represent the local state using a triple of type *store* × *uniqueId set* × *'any io*. The first component of the triple is the state of the procedure's local variables. The second component is a set of unique identifiers that are locally known. We use this to ensure that no unique identifiers can be created out of

thin air. Finally, the last component of the triple is an *io*-action representing the part of the procedure implementation that is still to be executed.

The procedure implementation must then be a function that takes such a triple and produces a *localAction*. To this end we define a function *toImpl* (see Figure 6.9). Besides the straight-forward translation of *io* actions to actions of the state machine, the definition of *toImpl* handles some additional aspects:

```
fun toImpl :: (('val store × uniqueId set × (('val,'op::{small,valueType},
'val::{small,valueType}) io)), 'op, 'val) procedureImpl where
  toImpl (store, knownUids, WaitLocalStep n) = (
      let (ok, store', n') = n store
      in LocalStep (ok ∧ (finite (dom store) ⟶ finite (dom (store'))))
                   (store', knownUids, n'))
| toImpl (store, knownUids, WaitBeginAtomic n) =
      BeginAtomic (store, knownUids, n)
| toImpl (store, knownUids, WaitEndAtomic n) =
      EndAtomic (store, knownUids,  n)
| toImpl (store, knownUids, WaitNewId P n) =
      NewId (λi. if P i then Some (store, knownUids ∪ uniqueIds i,  n i) else None)
| toImpl (store, knownUids, WaitDbOperation op n) = (
      if uniqueIds op ⊆ knownUids then
        DbOperation op (λr. (store, knownUids ∪ uniqueIds r, n r))
      else
        LocalStep False (store, knownUids, WaitDbOperation op n))
| toImpl (store, knownUids, WaitReturn v) = (
      if uniqueIds v ⊆ knownUids then
        Return v
      else
        LocalStep False (store, knownUids, WaitReturn v))
| toImpl (store, knownUids, Loop i body n) =
      LocalStep True
        (store, knownUids, (loop-body-from-V body i) ≫=io (λr.
          case r of Break x ⇒ n x
| Continue x ⇒ Loop x body n))
```

Figure 6.9.: *Definition of toImpl, which transforms an io-action into a state machine.*

1. The state of the local variables is threaded through the cases and updated for local steps.
2. We keep track of the unique identifiers that are locally known. In the case of a database operation or when a new unique identifier is generated, this set is updated accordingly.
3. When the procedure interacts with the outside, we check that only locally known unique identifiers are included in the output. If that is not the case, we produce an assertion error via *LocalStep*.
4. For executing loops, we unfold the loop for one iteration. This step to enter the loop is encoded as a separate *LocalStep* that has no other effect.

With this definition of *toImpl* we can now use our monadic language definition to implement our programs. Next, we will develop proof rules to verify programs written in this language.

6.2. Proof Rules

Proof rules are used to derive verification conditions in the form of mathematical formulas from a program. This can be done by traversing the program in execution order or in reverse order [GC10]. When using the reverse order, one starts with a post-condition for the program and then derives a necessary precondition. It is then checked whether the actual precondition implies the derived precondition. Since this is a check on a purely mathematical formula (i.e. it no longer contains *Io*-values), it is then possible to use an automatic solver. If the formula is first order and only uses supported constants, a tool like an SMT solver can be used. To ensure completeness, one usually computes the *weakest* precondition.

The corresponding approach for the forward direction is the strongest postcondition calculation. It starts from a given precondition of the program and then derives the strongest postcondition. It can then be checked that this derived postcondition implies the actually required postcondition.

In both cases, we generate a single formula to characterize the correctness of the program. The formulas derived using weakest preconditions is logically equivalent to the formula derived via strongest postcondition. However, the structure of the formula is different. Experiments by Grigore and others [Gri+09] have shown that this difference in the structure can have a significant effect on the running time of SMT solvers. Some programs are several times faster with one method compared to the other, but overall only a slight advantage for the weakest precondition approach was observed. This disadvantage of strongest postconditions is often attributed to additional existential quantifiers, which are required when calculating the strongest postcondition of assignments.

Instead of calculating the strongest postcondition as a single formula derived from the precondition, it is also possible to use symbolic execution. The main idea of symbolic execution is to represent the program state with symbolic variables. In particular, one symbolic variable is introduced for each input of the program. As a result, the aforementioned introduction of existential quantifiers at variable assignments is no longer necessary since every variable already has a symbolic value. More importantly though, further techniques become available when working on a symbolic state. A comparison by Kassios, Müller, and Schwerhoff [KMS12] shows that these techniques can result in a clear advantage of symbolic execution.

The most relevant techniques resulting in this advantage are the following:

1. The symbolic state can have more structure than a simple formula. In some cases this allows to perform checks directly on this structure without involving an SMT solver. It can also be used to simplify formulas,

resulting in verification conditions that are easier to process by SMT solvers.

2. Instead of invoking the solver only once, with the final formula, the solver can be invoked earlier. For example, in a branching construct, it can be invoked to check which branches are feasible. Also, program assertions can be checked immediately.

3. For handling branching constructs, symbolic execution allows for two variants. One option is to split the symbolic execution and check all branches separately. The other option is to split the execution only temporarily, and merge the symbolic states of the different branches again for the common statements following the branching constructs. In the worst case, the former can lead to an exponential number of paths that need to be executed. In return, the individual branches result in smaller formulas sent to the SMT solver.

For these benefits, we will formulate our proof rules in symbolic execution style. In particular, we want to use the symbolic state to generate simplified verification conditions. To this end, we keep track of some additional information in the symbolic state.

6.2.1. Symbolic State

There are two main goals behind defining a separate symbolic state and semantics on these states.

1. The state is organized such that it is easier to generate verification conditions. In particular, we exploit the structure of database updates in individual transactions to simplify the formulas for updating the database state. We also add more bookkeeping for unique identifiers, which allows us to automatically derive some properties about them from the state.

2. In the concrete state, the program control state (i.e. the remaining *io*-command to be executed) is part of the local state. We now separate the control state from the local variables, which is beneficial for reasoning about loop constructs. In loops, we want to formulate statements about the loop body in isolation, which is easier when the *io*-command is separated from the state.

The symbolic state is described by the record *proof-state* defined in Figure 6.10. It extends the record *invariant-context*, which includes the fields of the system state that can be used in invariants. This includes the history of database calls (*call*, *happensBefore*, *callOrigin*, *txOrigin*), the history of procedure invocations (*invocationOp*, *invocationRes*), and the set of unique identifiers known to clients (*knownIds*).

For the remaining parts of the system state (see Figure 5.1 on page 51), we include only the information of a single invocation. This invocation is given

record ('*proc*, '*any*, '*op*) *proof-state* = ('*proc*, '*op*, '*any*) *invContext* +
ps-i :: *invocId*
ps-generatedLocal :: *uniqueId set*
ps-generatedLocalPrivate :: *uniqueId set*
ps-localKnown :: *uniqueId set*
ps-vis :: *callId set*
ps-localCalls :: *callId list*
ps-tx :: *txId option*
ps-firstTx :: *bool*
ps-store :: '*any store*
ps-prog :: ('*proc*, ('*any store* × *uniqueId set* × ('*any*, '*op*, '*any*) *io*), '*op*, '*any*) *prog*

Figure 6.10.: *Definition of the symbolic state for symbolic execution. Extends the invariantContext record defined in Figure 5.1 on page 51.*

by the field *ps-i*. For the *visibleCalls* and *currentTransaction* of the current invocation, we use the fields *ps-vis* and *ps-tx*. Instead of the *generatedIds* from all invocations, we just store the local ones in *ps-generatedLocal*. The field *localState* is replaced in parts by the state of the mutable local variables in *ps-store*. The other part of the local state is the *io* action that is still to be executed. This is not included in the record, as it depends on additional type parameter for the result of the action.

For the purpose of generating simpler verification conditions, we include some additional ghost state, i.e. state that is not required for the symbolic execution but can be used to encode additional structure and invariants in the symbolic state.

For reasoning about unique identifiers, we include two ghost fields: The field *ps-generatedLocalPrivate* includes those identifiers that were locally generated and have not yet been exposed to other procedure invocations, which is by storing them in the database. The field *ps-localKnown* includes all identifiers that are locally known, which are the locally generated ones, the ones passed to the procedure invocation via parameters, as well as identifiers returned by database queries.

To simplify formulas related to database operations, we include two fields: First, the field *ps-localCalls* stores a list of calls performed in the current transaction in the order in which they were executed. This makes it easier to reason about updated database states without unfolding all updates of the happens-before relation. Second, the field *ps-firstTx* keeps track of whether the current invocation has already committed a transaction (value *False*) or is still in an earlier stage (value
True). This is useful to simplify the happens-before relation on invocations, which is simpler in the case of having only one transaction in an invocation.

Note that we have not explicitly included anything "symbolic" in our symbolic state. At this stage, this is not necessary, since we can simply use an Isabelle variable for symbolic state and Isabelle's logical premises to represent path conditions. We will make path conditions and symbolic variables explicit entities in Section 7.4, when we build a tool that works outside of Isabelle.

Relating Symbolic and Concrete States

To relate symbolic execution to our single-invocation semantics, we define a coupling relation between states of the former and states of a latter. The predicate *proof-state-rel PS CS* holds for a proof state of the symbolic execution *PS* and a concrete state *CS* from the single-invocation semantics, if and only if all the following conditions hold:

1. The concrete state is well-formed:

   state-wellFormed CS

2. The concrete and the symbolic state contain the same database calls:

   calls CS = calls PS

3. The happens-before relation in the concrete state is calculated from the happens-before relation of the symbolic state by adding the local calls of the current invocation (*ps-localCalls*) to the relation.

   happensBefore CS = updateHb (*happensBefore PS*) (*ps-vis PS*) (*ps-localCalls PS*)

4. The originating transaction of calls in the concrete state is the same as in the symbolic state, after updating the symbolic state with the local calls from the currently active transaction.

   callOrigin CS = map-update-all (*callOrigin PS*) (*ps-localCalls PS*) (*the* (*ps-tx PS*))

5. The originating procedure invocation for transactions is the same in both states, except for the current transaction which is not yet set in the symbolic state.

   transactionOrigin CS = (*case ps-tx PS of*
   Some tx ⇒ *transactionOrigin PS*(*tx* ↦ *ps-i PS*)
   | *None* ⇒ *transactionOrigin PS*)

   ∀ *t*. *ps-tx PS* ≜ *t* ⟶ *transactionOrigin PS t* = *None*

6. The unique identifiers known to clients are the same in both states.

   knownIds CS = (*knownIds PS*)

7. The history of procedure invocations is the same in both states.

   invocationOp CS = (*invocationOp PS*)

   invocationRes CS = (*invocationRes PS*)

8. The ghost state *ps-generatedLocal* in the symbolic state contains exactly the unique identifiers that where generated in the current invocation according to the concrete state.

   ps-generatedLocal PS = {*x*. *generatedIds CS x* ≜ *ps-i PS*}

9. The local state in the concrete state is a triple, where the first component is the store from the symbolic state, the second component is the locally known unique identifiers from the symbolic state, and the third component is some arbitrary *io*-action.

   $(\exists\, ps\text{-}ls.\ localState\ CS\ (ps\text{-}i\ PS) \triangleq (ps\text{-}store\ PS,\ ps\text{-}localKnown\ PS,\ \ ps\text{-}ls))$

10. The current procedure in the concrete state is the function *toImpl* defined in Figure 6.9.

    $currentProc\ CS\ (ps\text{-}i\ PS) \triangleq toImpl$

11. The visible calls in the concrete state is the union of the visible calls in the symbolic state with the local calls from the current transaction added.

    $visibleCalls\ CS\ (ps\text{-}i\ PS) \triangleq\ (ps\text{-}vis\ PS \cup set\ (ps\text{-}localCalls\ PS))$

12. The current transaction in the current invocation is the same for both states.

    $currentTransaction\ CS\ (ps\text{-}i\ PS) = ps\text{-}tx\ PS$

13. When no transaction is active in the symbolic state, then there are no uncommitted transactions in the concrete state.

    $(\forall\, tx'.\ ps\text{-}tx\ PS \neq Some\ tx' \longrightarrow transactionStatus\ CS\ tx' \neq Some\ Uncommitted)$

14. The local calls in the symbolic state contain exactly the calls from the currently active transaction in the concrete state.

    $(case\ ps\text{-}tx\ PS\ of$
    $\qquad Some\ tx' \Rightarrow set\ (ps\text{-}localCalls\ PS) = \{c.\ callOrigin\ CS\ c \triangleq tx'\}$
    $\quad \mid None \Rightarrow ps\text{-}localCalls\ PS = [])$

15. The list of local calls in the symbolic state is sorted by the happens-before relation of the concrete state.

    $(sorted\text{-}by\ (happensBefore\ CS)\ (ps\text{-}localCalls\ PS))$

16. The visible calls in the symbolic state and the calls from the currently active transaction are disjoint.

    $(ps\text{-}vis\ PS \cap set\ (ps\text{-}localCalls\ PS) = \{\})$

17. The calls from the currently active transaction in the symbolic state are not yet contained in the *callOrigin* map nor in the *happensBefore* relation.

    $(dom\ (callOrigin\ PS) \cap set\ (ps\text{-}localCalls\ PS) = \{\})$

    $(Field\ (happensBefore\ PS) \cap set\ (ps\text{-}localCalls\ PS) = \{\})$

18. The list *ps-localCalls* contains no duplicates.

    $distinct\ (ps\text{-}localCalls\ PS)$

19. The field *ps-firstTx* from the symbolic state is true, if and only if there is no non-empty transaction in the current invocation.

    (*ps-firstTx PS* ⟷ (∄ *c tx* . *callOrigin CS c* ≜ *tx* ∧ *transactionOrigin CS tx* ≜ *ps-i PS* ∧ *transactionStatus CS tx* ≜ *Committed*))

20. All committed calls from in the current invocation are contained in *ps-vis*.

    (∀ *c*. *i-callOriginI-h* (*callOrigin PS*) (*transactionOrigin PS*) *c* ≜ (*ps-i PS*) ⟶ *c* ∈ (*ps-vis PS*))

21. The locally generated unique identifers that are private are a subset of the locally generated unique identifiers and they only contain unique identifers that are private to the current procedure invocation.

    (*ps-generatedLocalPrivate PS* ⊆ *ps-generatedLocal PS*)

    (∀ *v*∈*ps-generatedLocalPrivate PS*. *uid-is-private* (*ps-i PS*) *CS v*)

22. The store for local mutable variables is finite. This ensures that we can always allocate new reference cells.

    (*finite* (*dom* (*ps-store PS*)))

23. The current invocation cannot generate unique identifiers out of thin air. Only the locally known unique identifiers can appear in outputs generated by the procedure.

    (*invocation-cannot-guess-ids* (*ps-localKnown PS*) (*ps-i PS*) *CS*)

24. All locally generated unique identifiers are locally known.

    (*ps-generatedLocal PS* ⊆ *ps-localKnown PS*)

25. Both states have the same program.

    prog CS = *ps-prog PS*

6.2.2. Symbolic Execution Semantics

Having defined the symbolic state, we now define the semantics of the symbolic execution. To this end we first define a predicate *step-io* that describes the state transitions possible for a single step in an symbolic execution. We use a relation instead of a function from state to state since the semantics is nondeterministic.

The relation has the form *step-io progInv qrySpec S cmd action S′ cmd′ Inv*. Here *progInv* is the program invariant and *qrySpec* is the database query specification of the program. The pre-state of the step is given by the *proof-state S* and *io*-command *cmd*. The post-state after the step is given by *S'* and *cmd'*. The final parameter *Inv* is a Boolean value which signals whether the step produced any errors. The full definition is shown in Figures 6.11 and 6.12. It is similar to the single-invocation semantics that we have introduced in Section 5.2.

```
step-io progInv qrySpec S cmd S' cmd' Inv ≡
(Inv ⟶ proof-state-wellFormed S')
∧ (case cmd of
  WaitReturn r ⇒ False
| WaitLocalStep cont ⇒
    (∃ store' ok.
       cont (ps-store S) = (ok, store', cmd')
     ∧ (Inv ⟷ ok ∧ finite (dom store'))
     ∧ (S' = S(|
        ps-store := store'
       |)))
| WaitBeginAtomic n ⇒
    ∃ t vis' calls' happensBefore' callOrigin' transactionOrigin' knownIds' invocationOp' invocationRes'.
      cmd' = n
    ∧ Inv
    ∧ ps-tx S = None
    ∧ progInv (invariantContext.truncate (S'(|transactionOrigin := transactionOrigin'|)))
    ∧ transactionOrigin' t = None
    ∧ (∀ t'. t' ≠ t ⟶ (transactionOrigin S t' ≜ ps-i S ⟷ transactionOrigin' t' ≜ ps-i S))
    ∧ chooseSnapshot-h vis' (ps-vis S) (λtx. Some Committed) callOrigin' happensBefore'
    ∧ consistentSnapshotH calls' happensBefore' callOrigin' (λtx. Some Committed) vis'
    ∧ (∀ c. callOrigin' c ≠ Some t)
    ∧ (S' = S(|
        calls := calls',
        happensBefore := happensBefore',
        callOrigin := callOrigin',
        transactionOrigin := transactionOrigin',
        knownIds := knownIds',
        invocationOp := invocationOp',
        invocationRes := invocationRes',
        ps-tx := Some t,
        ps-vis := vis'
      |))
    ∧ ps-growing S S' t
    ∧ transactionOrigin S t = None
| WaitEndAtomic n ⇒
    cmd' = n
  ∧ (S' = S(|
    happensBefore := updateHb (happensBefore S) (ps-vis S) (ps-localCalls S),
    callOrigin := map-update-all (callOrigin S) (ps-localCalls S) (the (ps-tx S)),
    transactionOrigin := transactionOrigin S(the (ps-tx S) ↦ ps-i S),
    ps-tx := None,
    ps-localCalls := [],
    ps-vis := ps-vis S ∪ set (ps-localCalls S),
    ps-firstTx := ps-firstTx S ∧ ps-localCalls S = []
  |))
  ∧ (Inv ⟷ progInv (invariantContext.truncate S'))
```

Figure 6.11.: *Semantics of single steps in symbolic execution (Part 1).*

```
| WaitNewId P n ⇒
  ∃ uidv uid.
    cmd' = n uidv
  ∧ Inv
  ∧ uniqueIds uidv = {uid}
  ∧ uid ∉ ps-generatedLocal S
  ∧ uid-fresh uid S
  ∧ P uidv
  ∧ (S' = S(|
       ps-localKnown := ps-localKnown S ∪ {uid},
       ps-generatedLocal := ps-generatedLocal S ∪ {uid},
       ps-generatedLocalPrivate := ps-generatedLocalPrivate S ∪ {uid}
      |))
| WaitDbOperation oper n ⇒
    (if Inv then
       ∃ c res.
        calls S c = None
      ∧ ps-tx S ≠ None
      ∧ uniqueIds oper ⊆ ps-localKnown S
      ∧ toplevel-spec qrySpec (current-operationContext S) (current-vis S) oper res
      ∧ cmd' = n res
      ∧ (S' = S(|
         ps-localKnown := ps-localKnown S ∪ uniqueIds res,
         ps-generatedLocalPrivate := ps-generatedLocalPrivate S − uniqueIds oper,
         calls := (calls S)(c ↦ Call oper res),
         ps-localCalls := ps-localCalls S @ [c]
        |))
     else
       S' = S ∧ cmd' = cmd ∧ ¬(uniqueIds oper ⊆ ps-localKnown S)
    )
| Loop i body n ⇒
    cmd' = loop-body-from-V body i ≫= (λr. case r of Continue x ⇒ Loop x body n |
Break x ⇒ n x )
    ∧ Inv
    ∧ (S' = S))
```

Figure 6.12.: *Semantics of single steps in symbolic execution (Part 2).*

Next, we extend this relation to multiple steps in the predicate *steps-io*. This predicate has the form *steps-io progInv qrySpec S cmd S' res*. The first parameters are equivalent to the case of a single step (*step-io*). However, instead of a new *io*-action and a flag to denote errors, we use a single value *res*. If the given *io*-action (*cmd*) completes without error, then *res* contains the returned value. If an error occurs during execution, then *res* is *None*. The full definition of *steps-io* is given in Figure 6.13.

inductive *steps-io* **for** *progInv qrySpec* **where**
steps-io-final:
steps-io progInv qrySpec S (*WaitReturn res*) *S* (*Some res*)
| *steps-io-error*:
step-io progInv qrySpec S cmd S' cmd' False
⟹ *steps-io progInv qrySpec S cmd S' None*
| *steps-io-step*:
⟦*step-io progInv qrySpec S cmd S' cmd' True*;
steps-io progInv qrySpec S' cmd' S'' res
⟧
⟹ *steps-io progInv qrySpec S cmd S'' res*

Figure 6.13.: *Symbolic execution of multiple steps (steps-io).*

We can use the predicate *steps-io* to define correctness of executing a command with respect to some postcondition P. The predicate *execution-s-check progInv qrySpec S cmd P* states, that executing the *io*-action *cmd* in a symbolic state S never produces any errors and after executing the complete command the postcondition P holds for the final state with the returned result. The formal definition is given in Figure 6.14.

definition
execution-s-check progInv qrySpec S cmd P ≡
∀ *S' res. steps-io progInv qrySpec S cmd S' res*
⟶ *proof-state-wellFormed S*
⟶ (*case res of Some r* ⇒ *P S' r* | *None* ⇒ *False*)

Figure 6.14.: *Definition of execution-s-check.*

definition
finalCheck Inv i S res ≡
Inv (*invContext.truncate* (*S*(|
invocRes := *invocRes S*(*i* ↦ *res*),
knownIds := *knownIds S* ∪ *uniqueIds res*|)))
∧ *uniqueIds res* ⊆ *ps-localKnown S*

Figure 6.15.: *Definition of finalCheck.*

When verifying a procedure, the postcondition is given by the predicate *finalCheck* defined in Figure 6.15. It checks that the program invariant holds

after the final state is updated with the returned result and that the result only contains unique identifiers known to the procedure invocation. However, the predicate can be different when checking parts of a procedure. For example, when checking a loop construct, the loop body is checked with a post condition that includes a loop invariant.

Linking symbolic execution and single-invocation semantics. Using the above definitions, we can now link symbolic executions with correctness in the single-invocation semantics. The corresponding Lemma is given in Figure 6.16. It states that a successful check using the symbolic execution (*execution-s-check*) implies that the program is correct with respect to the single-invocation semantics (*execution-s-correct*). The assumptions of the Lemma demand that the checked state S is an initial state of a procedure invocation which satisfies the program invariant. Also, the procedure implementation must be the function *toImpl* that we used to translate *io*-commands to the abstract state machine used in the semantics (see Figure 6.9 on page 96). Moreover, we demand that the program is well-formed with respect to its handling of unique identifiers, which ensures that it cannot produce unique identifiers out of thin air (this well-formedness property for programs is defined in Section 6.3).

Proof sketch. To prove this Lemma, we show that we can simulate steps taken in the single-invocation semantics using the symbolic execution as defined by *steps-io*. We show that if the single-invocation semantics produces any error, then so does the symbolic execution. The proof proceeds by an induction over the length of the trace in the single-invocation semantics. The predicate *proof-state-rel* that we introduced above is used to link the states of the two executions.

The case for the empty trace is trivial. For a nonempty trace we focus on the first step in the trace. We then distinguish the case where the current command is a *return* and the case for all other commands.

If the current command is a *return*, there are two possible actions according to the single-invocation semantics: Either the action is a *return* or there is an assertion failure because the implementation tried to return a value that contains invalid unique identifiers. Both cases are handled by the definition of *finalCheck*, which performs the same checks.

If the current command is not a *return*, we distinguish between the case where execution with the single-invocation semantics produces an error and where it does not. If it produces an error, we show that we can also produce an error with symbolic execution. Otherwise, we show that symbolic execution can perform a step that leads to a new symbolic state satisfying the *proof-state-rel* predicate with respect to the new state from the single-invocation semantics.

□

```
lemma execution-s-check-sound4:
  fixes S :: ('proc::valueType, 'any store × uniqueId set × ('any,'op, 'any)
impl-language-loops.io, 'op::valueType, 'any::valueType) state
  assumes a1: localState S i ≜ (Map.empty, uniqueIds op, ls)
    and a2: S ∈ initialStates' progr i
    and a3: currentProc S i ≜ toImpl
    and a4: invocOp S i ≜ op
    and prog-wf: program-wellFormed (prog S)
    and inv: invariant-all' S
    and qry-rel: crdt-spec-rel (querySpec progr) querySpec'
    and c: ⋀s-calls s-happensBefore s-callOrigin s-txOrigin s-knownIds s-invocOp
s-invocRes.
⟦⟦
⋀tx. s-txOrigin tx ≠ Some i;
invariant progr (|
  calls = s-calls,
  happensBefore = s-happensBefore,
  callOrigin = s-callOrigin,
  txOrigin = s-txOrigin,
  knownIds = s-knownIds,
  invocOp = s-invocOp(i↦op),
  invocRes = s-invocRes(i:=None)
|)
⟧⟧ ⟹
  execution-s-check (invariant progr) querySpec' (|
      calls = s-calls,
      happensBefore = s-happensBefore,
      callOrigin = s-callOrigin,
      txOrigin = s-txOrigin,
      knownIds = s-knownIds,
      invocOp = s-invocOp(i↦op),
      invocRes = s-invocRes(i:=None),
      ps-i = i,
      ps-generatedLocal = {},
      ps-generatedLocalPrivate = {},
      ps-localKnown = uniqueIds op,
      ps-vis = {},
      ps-localCalls = [],
      ps-tx = None,
      ps-firstTx = True,
      ps-store = Map.empty,
      ps-prog = progr|) ls (finalCheck (invariant progr) i)
shows execution-s-correct S i
```

Figure 6.16.: *Lemma describing the soundness of using execution-s-check for the verification of programs.*

6.2.3. Rule Definitions and Correctness

In the previous section, we have shown that we can use symbolic execution (*execution-s-check*) to show that a program is correct in the single-invocation semantics (*execution-s-correct*). In order to automate the symbolic execution, we now derive proof rules from the definition of *step-io*. We define one proof rule for each language construct, which allows the rules to be applied automatically based on the program structure. Another difference compared to directly using the definition of *steps-io* is that the rules instantiate some properties which allow Isabelle to do simplifications during symbolic execution. Moreover, the proof rules for loops include another inductive argument based on loop invariants.

Before defining the concrete rules, end we first prove a generic correctness criterion for proof rules, which is given in Figure 6.17.

```
lemma execution-s-check-proof-rule:
  assumes noReturn: ⋀r. cmd ≠ WaitReturn r
    and cont:
⋀PS' cmd' ok. ⟦
  step-io Inv crdtSpec PS cmd PS' cmd' ok;
  proof-state-wellFormed PS
⟧ ⟹
  ok
  ∧ (∃ res. cmd' = return res
    ∧ P PS' res)
  shows execution-s-check Inv crdtSpec PS  (cmd) P
```

Figure 6.17.: *Generic correctness criterion for proof rules.*

In general, our proof rules have the conclusion *execution-s-check Inv crdtSpec PS cmd P*, where *cmd* is a fixed command that is not a *return*. To show this conclusion, we have to consider all the steps that are possible from a well-formed pre-state *PS* to a post-state *PS'*. Here a symbolic state is called well-formed (predicate *proof-state-wellFormed*) if there is a concrete state related to the symbolic state according to *proof-state-rel*. For each of the possible steps, we then have to show that it does not produce any errors (*ok*) and that the predicate *P* holds for the new state and result of the command.

Below, we show how this generic rule is instantiated for the different kind of commands in our language.

Local Steps: References

Our language includes 3 different constructs to perform local steps interacting with the mutable local variable store via references: creating (*makeRef*), reading (*read*), and updating (*assign*). The proof rules for these constructs are shown in Figure 6.18. The proof rule follow the definition of the language constructs (see Figure 6.6 on page 94). However, unlike the original definitions we do not include a default action when working with invalid references. In-

stead, we add a precondition to the rules demanding that the used references are valid.

lemma *execution-s-check-makeRef*:
  assumes *cont*:
$\bigwedge ref.\ [\![$
  $ref = freshRef\ (dom\ (ps\text{-}store\ PS))$
$]\!] \Longrightarrow P$
    $(PS(\!|$
      $ps\text{-}store := (ps\text{-}store\ PS)(ref \mapsto intoAny\ a)$
     $|\!))$
    $(Ref\ ref)$

  shows *execution-s-check Inv crdtSpec PS* $(makeRef\ a)\ P$

lemma *execution-s-check-read*:
  assumes *validRef*: $iref\ r \in dom\ (ps\text{-}store\ PS)$
    and *cont*: $P\ PS\ (s\text{-}read\ (ps\text{-}store\ PS)\ r)$
shows *execution-s-check Inv crdtSpec PS* $(read\ r)\ P$

lemma *execution-s-check-assign*:
  assumes *validRef*: $iref\ r \in dom\ (ps\text{-}store\ PS)$
    and *cont*:
    $P\ (PS(\!|ps\text{-}store := ps\text{-}store\ PS(iref\ r \mapsto intoAny\ v)|\!))\ ()$
  shows *execution-s-check Inv crdtSpec PS* $(assign\ r\ v)\ P$

Figure 6.18.: *Proof rules for local steps involving references.*

Unique Identifier Creation

The rule for creating unique identifiers (*newId*) is shown in Figure 6.19. Remember that the *newId* command takes a predicate *tc*, which is a kind of type-check to determine what kind of unique identifier to create. In the rule, we demand this predicate to be true for an infinite set of identifiers. Since there can only be finitely many identifiers that already have been generated, this ensures that we can always succeed in finding a new one.

In the post-state we include the newly generated identifier in the set *ps-generatedLocalPrivate*. This allows us to later derive that the identifier is not already used in any other part of the system state. Section 6.3 explores this in detail.

Start of Transaction

The proof rule for starting a transaction (shown in Figure 6.20) is our largest proof rule. This is due to the fact that our technique handles interactions with other concurrently running procedure invocations in this rule using a nondeterministic state transition. In the rule, this state transition is modelled using new logical variables *s-calls'*, *s-happensBefore'*, *s-callOrigin'*, *s-transactionOrigin'*,

```
lemma execution-s-check-newId:
  assumes infPred: infinite (Collect tc)
    and cont: ⋀v vn. [[
    tc v;
    vn ∉ knownIds PS;
    vn ∉ ps-generatedLocal PS;
    uniqueIds v = {vn};
    uid-is-private-S (ps-i PS) PS vn
]] ⟹
    P
      (PS(| ps-localKnown := ps-localKnown PS ∪ {vn},
          ps-generatedLocal := ps-generatedLocal PS ∪ {vn},
          ps-generatedLocalPrivate := ps-generatedLocalPrivate PS ∪ {vn}
          |))
      v
shows execution-s-check Inv crdtSpec PS (newId tc) P
```

Figure 6.19.: *Proof rule for creating unique identifiers.*

s-knownIds', *vis'*, *s-invocationOp'*, and *s-invocationRes'* in the *cont* assumption.

For the new state we can assume the program invariant and that it is well-formed. We also can assume that the new state is reachable from the old state. This is formalized by the *ps-growing* predicate, which transfers the *state-monotonicGrowth* property we defined on concrete states to symbolic states. The full definition of the predicate is given in Figure 6.21.

Database Calls

The rule for database calls (*call*) is given in Figure 6.22. It includes two additional proof obligations: There must be a current transaction and the database operation must be shown to only include locally known unique identifiers.

In the *cont* part of the rule, we get to assume that the CRDT specification holds for the result of the database call in an execution context constructed from the current state.

In the rule, we also need to keep track of unique identifiers. The unique identifiers contained in the database operation can no longer be considered private to the current procedure invocation after writing them to the global database. Moreover, we can learn about new unique identifiers from the database result and thus we need to include the unique identifiers from the result in our set of locally known identifiers.

End of Transaction

The rule for committing a transaction is given in Figure 6.23. This rule introduced three proof obligations: First we need to show that there is an active transaction and secondly that at least one database call has been performed.

lemma *execution-s-check-beginAtomic*:
assumes *cont*: $\bigwedge tx\ s\text{-}calls'\ s\text{-}happensBefore'\ s\text{-}callOrigin'\ s\text{-}txOrigin'\ s\text{-}knownIds'\ vis'\ s\text{-}invocOp'\ s\text{-}invocRes'\ PS'.\ [\![$
$Inv\ (invContext.truncate\ PS')$;
$s\text{-}txOrigin'\ tx = None$;
$\bigwedge i\ op.\ invocOp\ PS\ i \triangleq op \Longrightarrow s\text{-}invocOp'\ i \triangleq op$;
$\bigwedge c.\ s\text{-}callOrigin'\ c \neq Some\ tx$;
$ps\text{-}vis\ PS \subseteq vis'$;
$vis' \subseteq dom\ s\text{-}calls'$;
$ps\text{-}firstTx\ PS \Longrightarrow (\bigwedge c\ tx.\ s\text{-}callOrigin'\ c \triangleq tx \Longrightarrow s\text{-}txOrigin'\ tx \neq Some\ (ps\text{-}i\ PS))$;
— consistency:
$causallyConsistent\ s\text{-}happensBefore'\ vis'$;
$transactionConsistent\text{-}atomic\ s\text{-}callOrigin'\ vis'$;
$\forall v \in ps\text{-}generatedLocalPrivate\ PS.\ uid\text{-}is\text{-}private'\ (ps\text{-}i\ PS)\ s\text{-}calls'\ s\text{-}invocOp'\ s\text{-}invocRes'\ s\text{-}knownIds'\ v$;
$PS' = (PS(\!|calls := s\text{-}calls',$
$happensBefore := s\text{-}happensBefore',$
$callOrigin := s\text{-}callOrigin',$
$txOrigin := s\text{-}txOrigin',$
$knownIds := s\text{-}knownIds',$
$invocOp := s\text{-}invocOp'(ps\text{-}i\ PS := invocOp\ PS\ (ps\text{-}i\ PS)),$
$invocRes := s\text{-}invocRes'(ps\text{-}i\ PS := None),$
$ps\text{-}vis := vis',$
$ps\text{-}localCalls := [],$
$ps\text{-}tx := Some\ tx$
$|\!|))$;
$proof\text{-}state\text{-}wellFormed\ PS'$;
$ps\text{-}growing\ PS\ PS'\ tx$
$]\!] \Longrightarrow$
$P\ PS'\ ()$
shows *execution-s-check Inv crdtSpec PS beginAtomic P*

Figure 6.20.: *Proof rule for starting a transaction.*

```
definition ps-growing S S' t ≡
  ∃ CS CS'.
    proof-state-rel S CS
  ∧ proof-state-rel S' CS'
  ∧ ps-localCalls S = []
  ∧ ps-localCalls S' = []
  ∧ ps-tx S = None
  ∧ ps-tx S' ≜ t
  ∧ ps-i S' = ps-i S
  ∧ ps-prog S' = ps-prog S
  ∧ ps-generatedLocal S' = ps-generatedLocal S
  ∧ ps-generatedLocalPrivate S' = ps-generatedLocalPrivate S
  ∧ ps-localKnown S' = ps-localKnown S
  ∧ ps-firstTx S' = ps-firstTx S
  ∧ ps-store S' = ps-store S
  ∧ state-monotonicGrowth (ps-i S) CS (CS' (|
        txStatus := (txStatus CS')(t := None),
        txOrigin := (txOrigin CS')(t := None),
        currentTx := (currentTx CS')(ps-i S := None),
        localState := (localState CS')(ps-i S := localState CS (ps-i S)),
        visibleCalls := (visibleCalls CS')(ps-i S := visibleCalls CS (ps-i S))|))
```

Figure 6.21.: *Definition of the ps-growing predicate.*

```
lemma execution-s-check-call:
  assumes in-tx: ps-tx PS ≜ tx
    and unique-wf: uniqueIds OP ⊆ ps-localKnown PS
    and cont: ⋀c res. [[
toplevel-spec crdtSpec (|
      calls = calls PS,
      happensBefore=updateHb (happensBefore PS) (ps-vis PS) (ps-localCalls PS)|)
      (ps-vis PS ∪ set (ps-localCalls PS))
       OP res;
Op PS c = None
]] ⟹
    P
      (PS(|calls := (calls PS)(c ↦ Call OP res),
        ps-generatedLocalPrivate := ps-generatedLocalPrivate PS − uniqueIds OP,
        ps-localKnown := ps-localKnown PS ∪ uniqueIds res,
        ps-localCalls := ps-localCalls PS @ [c]
       |))
      res
shows execution-s-check Inv crdtSpec PS (call OP) P
```

Figure 6.22.: *Proof rule for a database call.*

While the latter is not strictly necessary, we included it in the rule to avoid further case distinctions in the newly introduced assumptions.

The third, and most important, proof obligation in the rule is to show that the invariant holds for the new state after committing the transaction.

To simplify the proof of the invariant, we add a few assumptions in the *cont* case of the rule. These assumptions help the simplifier with typical expressions that might appear in invariants.

Sequential Composition

The rule for sequential composition with the bind ($\gg=$) operation is given in Figure 6.24. The idea is that we can check the initial command with the check for the continuation moved into the postcondition.

Invocation Return

In the case of a *return* command, we simply have to check the postcondition P with respect to the result and the final state, which we can assume to be well-formed. The corresponding rule is given in Figure 6.25.

Loops

To support loops, we use the classical approach of loop invariants. We first prove a rule for the general loop construct and then use this to derive proof rules for the higher-level loop constructs.

The general rule is given in Figure 6.26. Remember that the general *Loop* construct consists of an initial accumulator state, a loop body that takes the accumulator and returns a loop result. The loop result is either *Continue a* with a new accumulator or it is *Break r*, where r is a result passed to the continuation of the Loop construct.

For the rule, we assume that the continuation always has the form $\mathit{return} \circ f$ for some function f. The loop invariant is a predicate with three arguments: the symbolic state before executing the loop, the current accumulator, and the current symbolic state. In principle, it is not necessary to include the state before executing the loop in the invariant, since this state is known when applying the rule. However, if we want to annotate the program text with loop invariants, this information would not be available otherwise.

The proof rule for the loop construct then introduces two proof obligations: First, the loop invariant must be shown to hold for the initial state and the initial accumulator value. Second, it must be shown that the loop body is correct. This means that the loop invariant is maintained when the body returns with *Continue* and that the postcondition P holds when it returns with *Break*.

Unlike the previously introduced rules, the *Loop* rule is not suitable for automation, since the loop invariant is not fixed. To avoid this problem, we define annotated loop constructs with an additional parameter for the loop invariant. The loop invariant is not used in the definition and is only used

lemma *execution-s-check-endAtomic*:
assumes *in-tx*: *ps-tx PS* ≜ *tx*
and *tx-nonempty*: (*ps-localCalls PS*) ≠ []
and *cont*: ⋀*PS'*. [[
distinct (*ps-localCalls PS*);
⋀*c*. *c*∈*set* (*ps-localCalls PS*) ⟹ *callOrigin PS c* = *None*;
⋀*c*. *c*∈*set* (*ps-localCalls PS*) ⟹ *c* ∉ *ps-vis PS*;
⋀*c c'*. *c*∈*set* (*ps-localCalls PS*) ⟹ (*c*,*c'*) ∉ *happensBefore PS*;
⋀*c c'*. *c*∈*set* (*ps-localCalls PS*) ⟹ (*c'*,*c*) ∉ *happensBefore PS*;
invocation-happensBeforeH
(*i-callOriginI-h* (*map-update-all* (*callOrigin PS*) (*ps-localCalls PS*) *tx*) (*txOrigin PS*(*tx* ↦ *ps-i PS*)))
(*updateHb* (*happensBefore PS*) (*ps-vis PS*) (*ps-localCalls PS*))
=
(*if ps-firstTx PS then*
invocation-happensBeforeH (*i-callOriginI-h* (*callOrigin PS*) (*txOrigin PS*)) (*happensBefore PS*)
∪ {*i'* | *i' c'*. *c'* ∈ *ps-vis PS* ∧ *i-callOriginI-h* (*callOrigin PS*) (*txOrigin PS*) *c'* ≜ *i'*
∧ (∀ *c''*. *i-callOriginI-h* (*callOrigin PS*) (*txOrigin PS*) *c''* ≜ *i'* ⟶ *c''* ∈ *ps-vis PS*) } × {*ps-i PS*}
else
invocation-happensBeforeH (*i-callOriginI-h* (*callOrigin PS*) (*txOrigin PS*)) (*happensBefore PS*)
− {*ps-i PS*} × {*i'*. (*ps-i PS*,*i'*) ∈ *invocation-happensBeforeH* (*i-callOriginI-h* (*callOrigin PS*) (*txOrigin PS*)) (*happensBefore PS*) });
PS' = *PS*(|*happensBefore* := *updateHb* (*happensBefore PS*) (*ps-vis PS*) (*ps-localCalls PS*),
callOrigin := *map-update-all* (*callOrigin PS*) (*ps-localCalls PS*) *tx*,
txOrigin := *txOrigin PS*(*tx* ↦ *ps-i PS*),
ps-vis := *ps-vis PS* ∪ *set* (*ps-localCalls PS*),
ps-localCalls := [],
ps-firstTx := *False*,
ps-tx := *None*
|)

]] ⟹
Inv (*invContext.truncate PS'*)
∧ *P PS'* ()
shows *execution-s-check Inv crdtSpec PS endAtomic P*

Figure 6.23.: *Proof rule for committing a database transaction.*

lemma *execution-s-check-bind*:
assumes *execution-s-check progInv qrySpec S cmd* (λ*S' r*.
execution-s-check progInv qrySpec S' (*cont r*) *P*)
shows *execution-s-check progInv qrySpec S* (*cmd* ≫= *cont*) *P*

Figure 6.24.: *Proof rule for sequential composition via* ≫=.

lemma *show-execution-s-check-return*:
assumes *P PS r*
shows *execution-s-check Inv crdtSpec PS* (*return r*) *P*

Figure 6.25.: *Proof rule for returning from a procedure invocation.*

lemma *execution-s-check-Loop*:
fixes *init* :: *'any*::*valueType*
and *LoopInv* :: (*'b*::*valueType*, *'any*, *'op*::*valueType*) *proof-state* ⇒ *'any* ⇒ (*'b*, *'any*, *'op*) *proof-state* ⇒ *bool*
and *f* :: *'any* ⇒ *'a*
assumes
inv-pre: *LoopInv PS init PS*
and *cont*:
⋀*acc PSl*. ⟦
LoopInv PS acc PSl
⟧ ⟹ *execution-s-check Inv crdtSpec PSl* (*body acc*)
(λ*PS' r*. *case r of Break res* ⇒ *P PS'* (*f res*)
| *Continue acc'* ⇒ *LoopInv PS acc' PS'*)
shows *execution-s-check Inv crdtSpec PS* (*Loop init* (*loop-body-to-V body*) (*return* ∘ *f*)) *P*

Figure 6.26.: *Proof rule for the general loop construct.*

so that the corresponding proof rules can be applied automatically, without manual instantiation of variables.

We now present the proof rules for the higher level loop constructs. The rules are given in Figure 6.27.

The first rule is for the *loop* construct. This is similar to the generic *Loop*, but instead of using type *'any* for the accumulator and the return value of the loop, it allows to use any *countable* datatype.

In the second rule, we handle *while* loops, which are similar to the general looping construct but do not have a builtin accumulator nor a result value. Consequently, the loop invariant only takes the initial state and the current state as parameters. Overall, the rule is similar to the first rule.

Finally, the third rule in Figure 6.27 addresses the *forEach* loop. This loop iterates over a given list, producing a result for each element in the list. As typical loop invariants (see for example [Gri82]) involve statements about an already processed section and about the section of data that is still to be processed, we design the general form of the loop invariant accordingly. The loop invariant takes 5 parameters: The pre-state of the loop, the list of elements that already have been processed, the list of results that have been produced for these processed elements, the list of elements still to be processed, and finally the current state. This is similar to proof rules used in Spec# [Jac+05], where loop invariants can refer to the list of values that have already been returned by the loop iterator. Implicitly, the loop invariant includes the fact that the list of processed elements and the list of results have

lemma *execution-s-check-loop*:
  fixes *PS* :: ('*proc*::*valueType*, '*any*::{*small*,*valueType*,*natConvert*}, '*op*::*valueType*) *proof-state*
    and *init* :: '*acc*::*countable*
    and *body* :: '*acc* ⇒ (('*acc*, '*a*::*countable*) *loopResult*, '*op*, '*any*) *io*
  assumes *inv-pre*: *LoopInv PS init PS*
    and *cont*:
⋀*acc PSl*. ⟦
  LoopInv PS acc PSl
⟧ ⟹ *execution-s-check Inv crdtSpec PSl* (*body acc*)
    (λ*PS*′ *r*. *case r of Break x* ⇒ *P PS*′ *x*
                | *Continue x* ⇒ *LoopInv PS x PS*′)
  shows *execution-s-check Inv crdtSpec PS* (*loop-a init LoopInv body*) *P*

lemma *execution-s-check-while*:
  assumes *inv-pre*: *LoopInv PS PS*
    and *cont*:
⋀*PSl*. ⟦
  LoopInv PS PSl
⟧ ⟹ *execution-s-check Inv crdtSpec PSl body*
    (λ*PS*′ *r*. *if r then P PS*′ ()
          else LoopInv PS PS′)
shows *execution-s-check Inv crdtSpec PS* (*while-a LoopInv body*) *P*

lemma *execution-s-check-forEach*:
  assumes *inv-pre*: *LoopInv PS* [] [] *elems PS*
    and *iter*:
⋀*done results t todo PSl*. ⟦
  LoopInv PS done results (*t*#*todo*) *PSl*;
  length done = *length results*;
  elems = *done*@*t*#*todo*
⟧ ⟹ *execution-s-check Inv crdtSpec PSl* (*body t*) (λ*PS*′ *r*. *LoopInv PS* (*done*@[*t*])
(*results*@[*r*]) *todo PS*′)
    and *cont*:
    ⋀*results PSl*. ⟦
  LoopInv PS elems results [] *PSl*;
  length elems = *length results*
⟧ ⟹ *P PSl results*
shows *execution-s-check Inv crdtSpec PS* (*forEach-a elems LoopInv body*) *P*

Figure 6.27.: *Proof rule for different looping construct.*

the same length and that the list of processed elements concatenated with the list of elements to process is the list of all elements.

The rule then includes three proof obligations: First, we have to show that the invariant holds for the initial states with no elements processed yet. Next, we have to show that the loop body maintains the invariant. Here, we can assume that there is one more element to be processed. In the last proof obligation, we have to show that the loop invariant implies the postcondition P after all elements are processed.

Further Aspects of Loops. Our proof rules for loops provide the basic infrastructure to prove partial correctness of programs involving loops. For simplicity, we do not include a total correctness proof, which could be done by extending the proof rules with loop variants. These are expressions that must be shown to decrease in every loop iteration with respect to some well-founded ordering [Flo93].

Another aspect that we omitted for simplicity is loop framing. This means that we have to explicitly specify which parts of the state cannot change in the state. This problem can often be solved by statically determining the set of variables that are updated in a loop. For example in Dafny, this is done automatically and users can use special annotations to define which variables might change within a loop [FL17].

6.2.4. Combining Proof Rules with Eisbach

We now have defined proof rules for all constructs of our language. With the goal to automate verification of applications within Isabelle, we now define a custom tactic to derive verification conditions from a procedure implementation. To this end we use Eisbach [MMW16], which is a DSL for writing tactics within Isabelle.

A tactic in Isabelle is a function that takes an Isabelle proof state (an ordered list of goals to prove) and produces a lazy stream of new proof states. This stream represents alternative outcomes and can be used for backtracking. Tactics can be composed in different ways. It is possible to focus on certain subgoals and ignore others. By default, tactics work on the first subgoal only.

We quickly introduce here the composition operators that we use for our tactic (see Section 6.4.1 of the Isabelle Reference Manual [Wen+19]):

- The comma operator (“,”) composes two tactics sequentially. The second tactic is applied on all proof states produced by the first tactic in a depth-first fashion.
- The semicolon operator (“;”) is called structural composition. The left tactic focuses on the first subgoal and the right tactic is applied for each new subgoal produced by the first tactic.
- The question mark (“?”) tries to apply a tactic. If the tactic fails, i.e. it produces an empty stream, it instead produces a singleton stream containing the unchanged proof state.

- The plus operator ("+") repeats a tactic at least once and until it can no longer be applied.

The *method* construct offered by Eisbach allows to give a name and parameters to a tactic. It also allows the definition of recursive tactics.

```
method repliss-vcg-step1 uses asmUnfold =
  (rule repliss-proof-rules;
    ((subst(asm) asmUnfold)+)?;
     (intro impI conjI)?;
     clarsimp?;
     (intro impI conjI)?;
     (rule refl)?)

method repliss-vcg-step uses asmUnfold =
  (repliss-vcg-step1 asmUnfold: asmUnfold;
    (repliss-vcg-step asmUnfold: asmUnfold)?)

method repliss-vcg-l uses impl asmUnfold =
  ((simp add: impl)?,
    (unfold atomic-def skip-def)?,
    simp?,
    repliss-vcg-step asmUnfold: asmUnfold)
```

Figure 6.28.: *Eisbach tactics for symbolic execution.*

Using the constructs introduced above, we define our tactic for symbolic execution in Figure 6.28. The entry point is *repliss-vcg-l*, which takes two fact collections *impl* and *asmUnfold* as parameters. The parameter *impl* is supposed to contain the definitions of the procedure implementation. In the first step, we use these to simplify the goal, which results in a state where the program text of the current procedure is available. Next, we unfold the definitions of language constructs that are merely syntactic sugar for other constructs, since we do not have proof rules available for the sugared form. Then we try to simplify the goal again and invoke the tactic *repliss-vcg-step*. The tactic *repliss-vcg-step* is a recursive tactic that processes a single step using *repliss-vcg-step1* and then recursively tries to solve all new subgoals. Effectively, this strategy will remove all subgoals with an *execution-s-check* and only leave the other proof obligations to be solved.

For a single step, the tactic *repliss-vcg-step1* first tries to apply all the different proof rules. The collection *repliss-proof-rules* basically includes all the proof rules we introduced earlier. The only additional change is in the rule for *endAtomic*, which we restructure such that the proven invariant at the end of the transaction can be assumed to hold for proving the remaining postcondition *P*.

After applying the right proof rule, the new subgoals are tried to be simplified. To this end, we first unfold the definitions from the parameter *asmUnfold* in the assumptions of the new subgoals. This parameter is supposed to contain definitions that are useful to eliminate other proof obligations automatically

when unfolded. We then simplify the subgoals and split conjunctions into several goals. Finally, we try to use the rule *refl*, which eliminates schematic variables in simple equations. For example this is necessary when using the rule for database calls, since it contains the assumption *ps-tx PS* ≜ *tx* with a variable *tx* that is not directly bound by the rule application.

6.2.5. Example: Sum of List

We now show the use of the proof rules using a simple example including loops and references, but without any database interaction. This demonstrates that we have the basic functionality of a verifier for an imperative programming language.

The procedure shown in Figure 6.29 takes a list of natural numbers (*p-list*) and calculates the maximum element in the list. We formulate this correctness criterion as a program invariant in the definition *inv1*. It states that whenever the operation of an invocation *i* is *PMax list*, the list is nonempty, and the result of the invocation is *r*, then *r* is the maximum element in the list.

```
definition max-impl :: nat list ⇒ (val,operation,val) io where
max-impl p-list ≡
  do {
    resR ← makeRef 0;
    forEach-a p-list (loop-inv resR p-list) (λx. do {
      res ← read resR;
      resR :← max x res
    });
    res ← read resR;
    return (Nat res)
}

definition loop-inv :: nat ref ⇒ nat list ⇒ (proc, val, unit) proof-state ⇒ nat list ⇒
'a list ⇒ nat list ⇒ (proc, val, unit) proof-state ⇒ bool where
loop-inv resR p-list PS-old Done res Todo  PS ≡
 ∃ re.
        (iref resR) ∈ dom (ps-store PS)
      ∧ re ::= s-read (ps-store PS) resR
      ∧ (if Done = [] then re = 0 else re = Max (set Done))
      ∧ (only-store-changed PS-old PS)

definition inv1 where
inv1 op res ≡ ∀ i list r.
    op i ≜ PMax list
  ∧ list ≠ []
  ∧ res i ≜ r
  ⟶ (r = Nat (Max (set list)))
```

Figure 6.29.: *Example of a procedure to calculate the maximum of a list.*

To prove the correctness of the implementation, we need to annotate the

loop with an invariant, which is given in the definition of *loop-inv* in Figure 6.29. The essential part of the loop invariant states that the variable *resR* contains the maximum of the numbers processed so far or 0 if no element has been processed yet. Besides this essential part, we also need to specify that *resR* is a valid reference. Moreover, we need a kind of loop framing condition to state that invocation history and other parts of the state are not changed by the loop. For this we have defined the predicate *only-store-changed* which states that the local variable store is the only part of the state that may change between the old and the new state.

Using this invariant, we can use our Eisbach method *repliss-vcg-l* to generate verification conditions. The resulting verification conditions can then be discharged automatically by unfolding the definitions.

We also verified a variant of this example that uses a while-loop instead of a for-loop[4]. The corresponding proof is similar, but requires a bit more manual work. Since the proof rule does not include the idea of splitting the input list into a part that is already processed and a remaining part, this has to be done using existential quantifiers in the loop invariant. Moreover, the implementation requires references as there is no inherent loop state that can be used. Thus, the invariant need to address the references and state that they are valid and that there are no aliases.

6.3. Handling unique identifiers

The properties of unique identifiers often play a role in the verification of applications. However, we want to avoid proving these properties again for every program. In this section, we generalize these properties, which include statements about where unique identifiers may appear in the system state and history. This requires some well-formedness properties of programs, which we handle first.

6.3.1. Well-formedness of Programs

Our semantics from Section 5.1 already guarantees some basic properties for unique identifiers: The action for generating unique identifiers never generates an identifier that has been generated before and clients can never invoke a method using unique identifiers not exposed to them. However, the semantics does not prevent programs from constructing unique identifiers out of thin air, without using the identifier generation action and without obtaining the unique identifier from invocation arguments or database call results. Similarly, the semantics does not prevent CRDT specifications from generating new unique identifiers.

To efficiently keep track of where unique identifiers can be used, we need to exclude such behavior. We do this by defining the well-formedness of pro-

[4]The code for the while-loop example is available in the Isabelle theories under `example_loop_max.thy`.

grams. A program is well-formed, if its invocations and its database queries cannot guess unique identifiers:

program-wellFormed progr ≡
invocations-cannot-guess-ids progr ∧
queries-cannot-guess-ids (*querySpec progr*)

We first define the property for invocations and then for queries.

Since a program can use an arbitrary Isabelle function for the state machine implementation of procedures, we cannot use a syntactic definition. Instead, we base the definition on the semantics. First, we define inputs and outputs for actions in a trace. When an invocation learns about a new unique identifier, it is considered an input. This is the case when the procedure is invoked, when generating a unique identifier, and when getting a result from the database.

action-inputs (*ALocal ls*) = ∅
action-inputs (*ANewId i*) = *uniqueIds i*
action-inputs (*ABeginAtomic t c*) = ∅
action-inputs AEndAtomic = ∅
action-inputs (*ADbOp c opr res*) = *uniqueIds res*
action-inputs (*AInvoc proc*) = *uniqueIds proc*
action-inputs (*AReturn r*) = ∅
action-inputs ACrash = ∅
action-inputs (*AInvcheck b*) = ∅

Conversely, an action produces a unique identifier output, when information leaves the procedure invocation. This is the case when performing a database operation and when returning a value from the procedure:

action-outputs (*ALocal ls*) = ∅
action-outputs (*ANewId i*) = ∅
action-outputs (*ABeginAtomic t c*) = ∅
action-outputs AEndAtomic = ∅
action-outputs (*ADbOp c opr res*) = *uniqueIds opr*
action-outputs (*AInvoc proc*) = ∅
action-outputs (*AReturn r*) = *uniqueIds r*
action-outputs ACrash = ∅
action-outputs (*AInvcheck b*) = ∅

With the above definitions, we can define that an invocation cannot guess unique identifiers.

invocations-cannot-guess-ids progr =
(∀ *tr i a S′ uid*.
  initialState progr $\xrightarrow{tr\ @\ [(i,\ a)]}{}^{*}$ *S′* ∧ *uid* ∈ *action-outputs a* ⟶
  (∃ *a′*. (*i*, *a′*) ∈ *tr* ∧ *uid* ∈ *action-inputs a′*))

The definition states that if an action outputs a unique identifier, the same unique identifier must appear in the inputs of the same invocation earlier in the trace. Finally, the definition *invocations-cannot-guess-ids* specifies that the above property holds for all invocations.

Next, we define the similar property *queries-cannot-guess-ids* for database query specifications. This property states that if an unique identifier appears in the results of a database call but not in the operation arguments, then it must appear in an operation from the operation context.

queries-cannot-guess-ids qry =
(∀ *opr ctxt res x*.
  qry opr ctxt res ⟶
  x ∈ *uniqueIds res* ⟶
  x ∉ *uniqueIds opr* ⟶
  (∃ *cId opr res*. *calls ctxt cId* ≜ *Call opr res* ∧ *x* ∈ *uniqueIds opr*))

While the definition of a well-formed program is based on the semantics, it is not necessary to prove the well-formedness for each program. For database queries, we just have to prove well-formedness once for each CRDT. For procedure implementation we can ensure that the property holds by construction. In principle, there are two ways to do so. One is to design a static analysis that ensures no unique identifiers are generated. This could be done with abstract execution [CC77] or by using a deeply embedded language that does not provide any construct for generating unique identifiers out of thin air. The other method is to use dynamic checks by keeping track of the inputs and preventing each illegal output. This is the approach we chose for the Io-programs we introduced in Section 6.1.

Well-formedness of Io-Programs

Since we the definition of *toImpl* includes dynamic checks for the usage of unique identifiers, programs cannot guess unique identifiers if they are based on *toImpl* and the procedure implementation correctly captures the unique identifiers of the given invocations.This is captured by the following Lemma:

6.3.1.1 lemma *invocations-cannot-guess-ids-io*:
assumes *proc-initial*: ⋀*proc store localKnown cmd impl*.
  procedure progr proc = ((*store*, *localKnown*, *cmd*), *impl*) ⟹
    impl = *toImpl*
    ∧ *localKnown* = *uniqueIds proc*
shows *invocations-cannot-guess-ids progr*

For the proof, we first show that the set of locally known identifiers is always equal to the *trace-inputs* for each invocation. This is done by an induction over the steps taken. We then consider the actions that can add unique identifiers to the *trace-output* and show that because of the definition of *toImpl*, they can only add identifiers that are locally known.

6.3.2. Properties of private unique identifiers

In the proof rules (Section 6.2.3), we use the field *ps-generatedLocalPrivate* of the symbolic state, to keep track of locally generated unique identifiers. An new unique identifier is added to this set by the rule for the *newId* statement.

It is removed from the set, when it is exposed to other transactions via a database call. All identifiers in the set have the *uid-is-private'* property, which states that the identifier cannot be used in any database calls or procedure invocations in the history. Formally, it is defined as follows:

uid-is-private′ i s-calls s-invocOp s-invocRes s-knownIds uidv ≡
new-unique-not-in-invocOp s-invocOp uidv ∧
new-unique-not-in-calls s-calls uidv ∧
new-unique-not-in-calls-result s-calls uidv ∧
new-unique-not-in-invocRes s-invocRes uidv ∧ *uidv* ∉ *s-knownIds*

new-unique-not-in-invocOp iop uidv ≡
∀ *i op. iop i* ≜ *op* ⟶ *uidv* ∉ *uniqueIds op*

new-unique-not-in-calls iop uidv ≡
∀ *i op r. iop i* ≜ *Call op r* ⟶ *uidv* ∉ *uniqueIds op*

new-unique-not-in-calls-result iop uidv ≡
∀ *i op r. iop i* ≜ *Call op r* ⟶ *uidv* ∉ *uniqueIds r*

new-unique-not-in-invocRes ires uidv ≡
∀ *i r. ires i* ≜ *r* ⟶ *uidv* ∉ *uniqueIds r*

6.4. Completeness of Proof Rules

In this chapter, we derived rules for symbolic execution with respect to the single-invocation semantics. The challenges concerning completeness in this step are in the definition of *steps-io* and the definition of the proof rules. In particular the rules for loop constructs are interesting here, since they must allow to formulate a sufficiently strong loop invariant to be complete.

Since we have formulated the proof rules as theorems, it is not possible to prove completeness of the rules directly in Isabelle[5]. Thus, we only give an informal argument here. For the rules that are not handling loops, we just have to check that all preconditions available in the single-invocation semantics and the *steps-io* function are preserved in the rules. In fact, we even added more assumptions to the rules to help with automation.

For the while-loop rule, we used the standard technique of loop invariants, for which completeness has already been shown informally [Coo78] and formally in Isabelle/HOL [Nip06]. The proof rules for the other loops follow a similar pattern. When translated to a while-loop, the loop variable used in these constructs would need to be translated to a reference value. As the loop invariant allows to address the state of the loop variables, both cases would allow formulating equivalent loop invariants. Thus, the proof of completeness for while-loops should be transferable to the other loop constructs as well.

Another aspect of completeness of loop constructs is, whether it is always possible to annotate a program with sufficiently strong loop invariants – the

[5] We could have used an inductive definition for the rules, or we could prove completeness by separately showing that an inversion of the implication in the Lemma also is true.

completeness proofs referenced above only consider the case where the loop invariant is chosen dynamically with all quantified variables available. This can be a problem, if proof rules introduce new variables that are not available statically, when annotating the program. For example the *LOCAL* construct in Nipkow's Abstract Hoare Logics [Nip06] models local variables with shadowing and the corresponding Hoare rule introduces a new variable. As a consequence, it is not always possible to annotate a loop in the scope of a local variable ahead of time. In our language we do not have a similar construct for local variables, but we do have rules that introduce new variables.

One strategy to mitigate the annotation problem is to add constructs to refer to previous states. For example Simpl [Sch08] includes a special construct (*ANNO* and *FIX*) to introduce new logical variables, which can be used to relate old and new states. In Dafny [FL17; Lei20] there is a special expression *old* which can refer to the state at the beginning of the procedure or at a label in the program.

We include a similar construct in our rule for loops, which allows the loop invariant to refer to the old state before the start of the loop. The example of computing the sum of a list (Section 6.2.5) showed that this feature was useful to specify the parts of the state that are not changed by the loop. However, we have not further investigated the question whether this construct is necessary for completeness and whether it is sufficient.

Chapter 7

Design and Implementation of Repliss

We have developed the Repliss tool in order to evaluate the feasibility of our proof technique in an automated verification tool. With Repliss, it is possible to use the technique without knowledge of Isabelle. The sources for the Repliss tool are available on GitHub[1].

The input for the tool is given in the Repliss language. The language includes constructs for specifying the schema of the persistent data using replicated data types. To this end, the CRDTs presented in Section 4.1 are included in the language. The implementation of procedures is done in a simple imperative language, which is similar to the language we embedded in Isabelle earlier in Section 6.1. However, the language does not require the use of the *'any* type we used previously and thus more problems can be detected by the type checker. The language also supports a simple form of user-defined datatypes and it supports specifications using first order formulas.

Figure 7.1 gives a high level overview over Repliss' main components. The language frontend consists of a parser and a type-checker. We present the syntax and type rules of the language in Section 7.1. We then have two main

[1] `https://github.com/peterzeller/repliss`

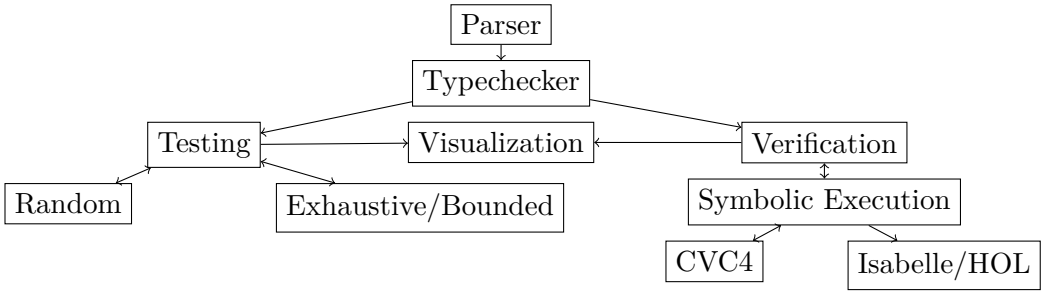


Figure 7.1.: *Overview over Repliss Components and data flow between components.*

components to check correctness of a Repliss application: Automated testing for finding bugs and verification for proving the absence of bugs.

We present our techniques for automated testing in Section 7.3. The techniques include two basic strategies. One based on random testing and one based on exhaustive bounded testing, where all executions within certain bounds are systematically checked. For verification purposes we use symbolic execution, which we present in Section 7.4. Both techniques share a common visualization component, which is used to show counter examples to the user.

7.1. Language

We begin by presenting the Repliss language. To present the syntax of the language, we use the ANTLR4 [Par13] grammar which we also use to generate the parser for the language. ANTLR4 grammars are similar to EBNF [Sta96] notation. Non-terminal symbols are written in lowercase and terminal symbols are written in quotes or upper case letters. Quoted terminal symbols are interpreted literally and upper case terminal symbols refer to tokens defined using regular expressions. Below we use `ID` for identifiers (`[a-zA-Z][a-zA-Z_0-9]*`) and `INT` for numbers (`'0'|[1-9][0-9]*`). Repetition is denoted by a star (`*`), at-least-once repetition by a plus (`+`) sign. Optional elements are marked with a question mark (`?`) and parenthesis are used for grouping. Alternatives are separated by vertical bars (`|`). Ambiguities in the grammar are resolved by precedence, where the first rule in an alternative has precedence over the later alternative. Moreover, operators are associated left to right by default, which can be overwritten with the option `<assoc=right>`.

Lexical Syntax Comments follow the C-style syntax (`//` for line comments and `/*...*/` for multiline comments). A newline symbol (`NL`) stands for one or more linebreaks and is used as a delimiter for most language constructs. A newline symbol is omitted when one of the following tokens is used at the end of the line or as the first non-whitespace character of the next line: `+`, `*`, `-`, `/`, `%`, `&&`, `||`, `:`, `::`, `=`, `==`, `!=`, `|`, `==>`, `<==>`, or a comma. Moreover, a newline symbol is omitted if the line ends with an open parenthesis[2] or the next line starts with a closing parenthesis, a dot, or a negation symbol.

The syntax for procedures uses indentation to delimit blocks. The tokens `STARTBLOCK` and `ENDBLOCK` are emitted by the lexer when the indentation is increased or decreased by one.

Programs Figure 7.2 shows the syntax for Repliss programs and the most important declarations. A program consists of a list of arbitrary declarations followed and ends with the end-of-file marker (`EOF`).

Type Definitions The rule `typedecl` describes the Syntax for type definitions. It combines the format for three different kind of types. There are types for

[2] This includes round parenthesis `()`, square brackets `[]`, and curly braces `{}`.

```
program: NL? (declaration NL?)* NL* EOF;

declaration:
      procedure
    | typedecl
    | operationDecl
    | queryDecl
    | axiomDecl
    | invariant
    | crdtDecl;

typedecl: ('idtype'|'type') ID typeParams? ('=' dataTypeCase ('|'
    dataTypeCase)*)?;

typeParams: '[' typeParam (',' typeParam)* ']';

typeParam: ID;

dataTypeCase: ID ('(' (variable (',' variable)*)? ')')?;

invariant: 'free'? 'invariant' (ID ':')? expr;

crdtDecl: 'crdt' keyDecl;

keyDecl: ID ':' crdttype;

crdttype:
      structcrdt
    | crdt;

structcrdt: ID? '{' keyDecl (',' keyDecl)* '}';

crdt: ID ('[' crdttype (','crdttype )* ']')?;

operationDecl: 'operation' ID '(' (variable (',' variable)*)? ')';

queryDecl: ('@inline')? 'query' ID '(' (variable (',' variable)*)? '
    )' ':' type
    ('=' expr | 'ensures' expr)?;

type: ID ('[' type (',' type)* ']')?;

procedure: 'def' ID '(' (variable (',' variable)*)? ')' (':' type)?
    stmt;

variable: ID (':' type)?;
```

Figure 7.2.: *Syntactical structure of Repliss progras*

unique identifiers (`idtype`), opaque types, and data types. Only data types can have a list of cases. Opaque types can be used for types like `String`, which are not included in Repliss and which do not need to be manipulated by the Repliss program. Id-types are types for which values can only be generated by Repliss programs (see `new` statement discussed later) and not by the environment. This provides some guarantees about how clients can use the API of the Repliss program, since they can only pass id-values as arguments if they have been previously exposed to a client.

Invariants The main mechanism for specifying applications is via invariants (the other mechanism being assertions, which we discuss below). An invariant is introduced with the keyword `invariant` and comprises a boolean expression. It can be marked as `free` in which case it is not checked but still can be used to prove the correctness of other invariants. This can be useful for debugging or for complicated invariants that cannot yet be checked by Repliss but can be verified externally.

Database Schema Conceptually, the database schema consists of a single struct CRDT (c.f. Section 4.1). In the syntax each field of the outermost struct is defined with the `crdt` keyword followed by a `keyDecl`, which consists of a name for the field and its CRDT type.

A CRDT type can either be a struct CRDT, which we write as a list of `keyDecl` in curly braces, or it can be a builtin CRDT type. Those types are used by their name and can take additional CRDT types or normal types as type arguments.

It is also possible to define custom operations and queries, if a type is not already provided by Repliss. However, these custom types cannot be composed with the existing maps as we cannot ensure that specifications have the expected format. Operations are introduced by the keyword `operation` and queries by the keyword `query`. A query can either be specified with an implementation given by a boolean formula (using `=` syntax) or using a post-condition (using `ensures` syntax). The post-condition can refer to the special variable `result` to specify which results are valid.

Procedures A procedure is introduced by the `def` keyword. The parameters must have an explicitly given type which is checked separately from the grammar. When the return type is not given, the procedure returns no value. The body of the procedure is a single statement, which is usually a block statement (see below).

Statements Figure 7.3 shows the Syntax for statements in Repliss. The first 6 cases comprise common language constructs of imperative languages: block statements, local variables, assignments, the conditional statements `if` and `match`, and a return statement. The `match`-statement provides pattern matching which can be used together with data types. The only supported patterns are variables that are bound while matching and datatype constructors.

```
stmt:
      blockStmt
    | localVar
    | assignment
    | ifStmt
    | matchStmt
    | returnStmt
    | newIdStmt
    | atomicStmt
    | crdtCall
    | assertStmt
    ;

blockStmt: STARTBLOCK stmt* ENDBLOCK;

localVar: 'var' variable ('=' expr)? NL;

assignment: ID '=' expr NL;

ifStmt: 'if' expr NL stmt ('else' NL? stmt)?;

matchStmt: expr 'match' NL STARTBLOCK matchCase* ENDBLOCK;

matchCase: 'case' expr '=>' NL STARTBLOCK stmt* ENDBLOCK;

returnStmt: 'return' expr (assertStmt)*  NL;

atomicStmt: 'atomic' NL stmt;

crdtCall: 'call' functionCall NL;

newIdStmt: ID '=' 'new' ID  NL;

assertStmt: 'assert' expr NL;
```

Figure 7.3.: *Syntax for statements in Repliss.*

```
expr:
      ID
    | ('true'|'false')
    | INT
    | expr '.' ID
    | '!' expr
    | expr 'is' 'visible'
    | expr 'happened' ('before'|'after') expr
    | expr ('*'|'/'|'%') expr
    | expr ('+'|'-') expr
    | expr ('<'|'<='|'>'|'>=') expr
    | expr ('=='|'!=') expr
    | expr '&&' expr
    | expr '||' expr
    | <assoc=right> expr '==>' expr
    | expr '<==>' expr
    | quantifierExpr
    | 'forall valid snapshots' '::' expr
    | functionCall
    | '(' expr ')'
    ;

quantifierExpr: ('forall'|'exists') variable (',' variable)* '::'
    expr;

functionCall: ID '(' (expr (',' expr)*)? ')';
```

Figure 7.4.: *Syntax for expressions in Repliss*

There are two statements for interacting with the database. The atomic statement executes a statement in the context of a transaction and the call statement performs an update operation on the database. When a call statement is used outside a transactional context, it is implicitly surrounded by an atomic block containing only the single call.

The `newIdStmt` is used to create a new unique identifier. The name after the **`new`** keyword refers to a type defined as **`idtype`**.

The assert statement is used to add additional checks to a procedure implementation. It is verified that the expression of an assert statement always evaluates to true when the assertion is executed. Just like all other expressions in programs, assertions can only access the local state of the current procedure invocation.

Expressions The syntax for expressions is given in Figure 7.4. The expressions include the standard constructs: There are variables (`ID`) and constants for booleans (`true`, `false`) and integers (`INT`). The operators on booleans are negation (`!`), conjunction (`&&`), disjunction (`||`), implication (`==>`), and equivalence (`<==>`). The general comparison operators (`==` and `!=`) can be used with all types. For integers, there are comparisons (`<`, `<=`, `>`, `>=`), addition (`+`), multiplication (`*`), subtraction (`-`), integer division (`/`), and modulo (`%`). The

semantics for division and modulo follow the Euclidean definition [Bou92], which is also used in the SMT-LIB standard [BFT16].

The language also includes universal (`forall`) and existential (`exists`) quantifiers. The quantifiers range over all values belonging to the type of the variable. Note that the set of values can change over time, for example new database calls, new procedure invocations or new unique identifiers can be created.

There are a few expressions specific to our domain, namely expressions to address the database and procedure invocation history. The expression `c is visible` checks whether a database call `c` is visible in the current context. The expressions `e1 happened before e2` checks whether a database call `e1` happened before a database call `e2`. The happens-before relation is the irreflexive, transitive relation tracking potential causality, as described in Section 5.1. The happens-before expressions is overloaded and also works when checking the happens-before relation of two procedure invocations. A procedure invocation i_1 is defined to be happened before a procedure invocation i_2 if both invocations contain at least one database call and all database calls in invocation i_1 happened before every database call in i_2.

To access information about history events, we use the property access syntax with a dot (`e.p`). The following properties `p` are supported:

op The operation performed by a database call.

info Information about the procedure and arguments of a procedure invocation event.

result The result returned by a procedure invocation. The value is `NoResult` if the invocation has not yet finished.

origin The procedure invocation in which a transaction or database call was executed.

transaction The transaction in which a database call was executed.

Expressions can also include function calls, which can be calls to datatype constructors or database queries. Queries are executed in the current context. The special quantifier `forall valid snapshots` can be used to create an implicit context, which allows queries to be used in invariants. Each invariant that uses queries outside this quantifier is implicitly enclosed by the quantifier.

There are some restrictions about which expressions can be used in invariants, assertions, and in program code.

1. Quantifiers can only be used in invariants and assertions.

2. Expressions concerning the history can only be used in invariants. There are the datatypes `InvocationId` and `CallId` and the expressions working with these types.

7.2. CRDT Library

In Chapter 4 we already introduced the different CRDTs and their semantics. Most of these CRDTs are also available in Repliss. Here we only present the API of the available datatypes and refer to Chapter 4 for the semantics.

Flags. The following variants of flags are supported: `Flag_ew`, `Flag_sew`, `Flag_dw`, and `Flag_sdw`. A flag provides the operations `Enable` and `Disable` and the query `ReadFlag: Boolean`.

Sets. Similar to flags, sets also come in 4 variants: `Set_aw[T]`, `Set_saw[T]`, `Set_rw[T]`, and `Set_srw[T]`. A set provides the operations `Add(x: T)`, `Remove(x: T)` and the queries `Contains(x: T): Boolean` and `GetSize: Int`. Since the specification for `GetSize` requires aggregates, the query is not supported by automatic verification.

Maps. Maps come in two variants: `Map_uw[K, V]`, `Map_dw[K, V]`. Here, `K` is an ordinary type for keys in the map and `V` is another CRDT type for the values. To perform operations on values, the map API provides the operation `MapOp(key: K, op: V.op)` and the query `MapQuery(key: K, qry: V.qry)`. The types `V.op` and `V.qry` refer to the operation and query types of the nested CRDT `V`. Moreover, there is an operation `DeleteKey(k: K)` and a query `ContainsKey(k: K): Boolean`.

Registers. The type `Register[T]` stores values of type `T`. The only operation is `Assign(x: T)` and the only query is `ReadRegister: T`. The register guarantees that the returned value is one of the latest assigned values, but it does not specify a strategy to choose from these values.

Multi-value Registers. The type `MultiValueRegister[T]` also provides an operation `Assign(x: T)`. For reading there is the query `ReadFirst: T`, which reads one the latest assigned values, just like the `Register` above. In addition, it is possible to use the query `MvContains(x: T): Boolean` that checks whether `x` is one of the latest assigned values.

Reading the complete set of values is not supported since this would require support for list or set values in Repliss.

Counter. The `Counter` type provides an operation `Increment(amount: Int)` and a query `GetCount: Int`. Since the specification requires aggregates, the query is not supported by automatic verification.

7.3. Automatic Testing

Repliss supports automated testing in order to find bugs in applications. This can be seen as a more lightweight method compared to verification, but it can

also simplify the verification process. When trying to verify an invariant that does not hold, it can be very helpful to be presented with a genuine counter example found by automatic testing. The examples are genuine in the sense that they show complete executions starting from the initial state. In contrast to this, the verification process can produce counter examples that start from a state that satisfies the invariant but is actually unreachable.

In order to support automated testing, we need a technique for executing specifications, which we discuss in Section 7.3.1. We then present different strategies for exploring executions: Randomized executions in Section 7.3.2 and systematic exploration of small executions in Section 7.3.4.

7.3.1. Executing Specifications

One problem in executing the system model described above, is that it requires evaluating first order logical formulas for checking properties and for determining the results of database queries, which can also be specified by the user. Validity of first order formula is undecidable in general [Chu36; Tur36]. However, we are interested in a simpler problem, which is to evaluate a first order formula on a given structure. Without restrictions, this problem is still undecidable. For example if we allow quantification over integers, one can express the fact that a diophantine equation has a solution, which is known to be undecidable [Mat70].

We address this problem by changing the semantics of unbounded quantifiers during evaluation. When quantifying over a non-finite type, we restrict the number of elements to a certain number N. In theory this means that we might miss some bugs in the case of universal quantification in specifications and that we might mistakenly report non-genuine bugs when existential quantifiers are used in specifications. While the former is not a big problem in testing, the latter is more concerning. However, the only unbounded types in Repliss are integers and user-defined datatypes that are recursive or include other unbounded types. Identifier types like `CallId` are unbounded, but quantifiers only range over the identifiers that have been created so far, which is a finite set. When a counter example is found related to unbounded quantification, we could generate a warning that the counter example might be non-genuine.

Using the above restrictions on quantifiers, we can evaluate them by checking all possible values. As an optimization, we use an approach based on narrowing.

Evaluating Quantifiers with Narrowing. Specifications often contain quantifiers that do not need to be checked for all elements. As an example consider the first invariant of our chat application from Section 2.1:

```
invariant forall g: InvocationId, m: MessageId, author: UserId,
    content: String ::
      g.info == getMessage(m)
  && g.result == getMessage_res(found(author, content))
  ==> (exists s: InvocationId, content2: String ::
        s.info == sendMessage(author, content2))
```

Here, the variables of `m`, `author`, and `content` depend on the chosen invocation `g`. We use a Narrowing approach inspired by Isabelle's Quickcheck [Bul12] or Lazy SmallCheck for Haskell [RNL08]. The main idea is to initially check a quantified formula with an abstract value. Only when this value is actually used, it will be refined (or narrowed) into more precise values. The aforementioned tools use Haskell's lazy evaluation and imprecise exceptions to detect the use of a value. When the corresponding exception is handled, the evaluation of the quantified formula is repeated with refined versions of the abstract value causing the exception.

Instead of exceptions, we use mutable state. When an abstract value needs to be evaluated to a concrete one, we simply reassign the variable with the first refined value and store the remaining refinements in a separate mutable variable. This avoids repeating the same computation.

We focus here on the optimization of equality checks, for which we use two abstract values:

1. `SymAny` is the abstract value representing all possible values.
2. `SymNotEq(x)` is an abstract value representing all possible values not equal to `x`.

When we execute an equation $x = e$ or $e = x$, where x is a variable and e an arbitrary expression, we evaluate e to a concrete value v_e. We then lookup the current value of x.

- If the value is concrete, we compare the two concrete values.
- If it is `SymAny`, we replace it with v_e, add `SymNotEq(`v_e`)` to the list of values to check for v (unless v_e is the only valid value for variable x), and then return *true* for the evaluation of the expression.
- If the value of x is `SymNotEq(`v_e`)`, the expression evaluated to *false*.
- Otherwise, we fall back to concrete values and check x with all possible remaining values. For `SymNotEq(`a`)` these are all values not equal to a.

When an evaluated expression has the form $C_1(\overline{a_1}) = C_2(\overline{a_2})$ for datatype constructors C_1 and C_2, we compare the constructors. If they are equal, we recursively check equality pairwise on the arguments $\overline{a_1}$ and $\overline{a_2}$. Similarly, if only one side is a datatype construction, we evaluate the other side and then continue as in the former case.

This simple evaluation strategy can be extended with additional abstract values and handled cases. For example, it would be possible to handle expressions of the form $x \in S$, such that x is only checked for the concrete elements in the set and one abstract value representing values not in the set. Another extension would be an abstract value for datatype values, where the constructor is concrete, but the arguments may be symbolic.

More advanced strategies for evaluating invariants might be possible as well. For example, we could use logical equivalence to rewrite formulas. It might

also be possible to make the evaluation of invariants incremental. Many components of the system state, in particular the history, grows monotonically, which could be exploited to only evaluate quantifiers on parts of the state that have changed. Another possible option would be to use SMT solvers to evaluate quantifiers, although these tools have been built and optimized for checking satisfiability and not for evaluating a formula on a given structure. As such, they do not provide an API to define a structure and it would be necessary to define it through custom type definitions and logical formulas. This encoding might already be more expensive than the approach we are using.

7.3.2. Random Executions

Our first (naive) approach of randomly exploring the space of concurrent executions failed to find invariant violations for our examples. We had to introduce heuristics to guide the search towards interesting cases of concurrency. These heuristics are a direct consequence of the following observations:

1. Bugs happen more often, when there are few objects with many concurrent accesses.
2. When a procedure consists of several transactions, bugs often appear when many changes are pulled in between transactions.
3. Causal consistency tends to tame the chaos introduced by randomness, so a random choice must consider causality.
4. Most bugs can be triggered by a short sequence of actions. Longer sequences of random actions can lead to save states, where no bugs can appear (for example, deleted objects and final state of a state-machine are often safe and cannot become unsafe again).

Point 1 was the easiest to address. We simply limit the size of the domain. For our examples a size of 3 values per primitive type worked well (e.g. we only use 3 different strings in executions). For id-types, we inspect the results of procedures and when a procedure has generated more than 3 unique identifiers, we stop generating new calls to the procedure.

Points 2-4 were not addressed well in our initial approach, where we simply selected a random subset of all transactions as the set of pulled transactions. Because all causal dependencies are included in a pull, picking a transaction with many dependencies, pulled in almost all transactions, which led to an almost linear history and did not reveal many bugs. Furthermore, it was unlikely that the next transaction in the same invocation would start on a substantially different snapshot. We addressed these issues with the following approach: First we calculate the set of transactions, which are not yet visible in the current invocation. Then we either pick one or two random transactions from this set, which simulates pulling in changes from one or two other replicas. This addresses point 2, since we always pull in new transactions when possible (an exception is the first pull in an invocation, where we allow starting from

the empty state). To address point 3 and 4, we introduced a bias towards older transactions, so that we avoid including a big set of causal dependencies.

7.3.3. Shrinking Counter Examples

When we find an invariant violation, we try to shrink the execution in order to present a small counter example to the user. An execution is defined by the trace of actions, which were randomly generated before. The trace includes all nondeterministic choices made by the system and therefore a trace can be executed deterministically. This property is important for replaying traces.

To allow efficient shrinking of the trace, the execution also has to be stable when removing some actions from the trace: this should only have minimal effects on other actions. We achieve this with the following two design decisions: First, we fix all generated identifiers in the actions of the trace, so that identifiers are not affected by removing previous actions. Then, we ensure that actions can still be executed, even when the context has changed. For example, a `StartTransaction` action can include pulled transactions, which have already been removed, and we simply ignore them. Moreover, the set of pulled transactions stored in the action already includes all causal dependencies, so removing one transaction has a minimal effect on the overall set of pulled transactions.

When we encounter an invalid action during execution, we simply ignore it and report the invalid action to the shrinking process. That way, we can directly remove all actions that have been invalidated by removing a single action. For example, when we remove the call to a procedure that returns a new unique identifier, this approach removes all actions in that call, as well as all calls using the generated identifier.

The shrinking algorithm itself is then straightforward: We try removing an action from the current trace, starting with the first action. When the reduced trace still triggers the invariant violation, we continue shrinking the reduced trace. Otherwise, we try to remove the next action from the trace instead and continue as above.

7.3.4. Systematic Execution

Besides the random executions with shrinking described above, we also implemented a systematic exploration of small executions. This is similar to the strategy used by SmallCheck [RNL08]. The idea is similar to random executions. However, instead of using randomness for picking a value at points of choice, we systematically enumerate all values. This process starts with small values, which means that discovered counter examples are guaranteed to be minimal (with respect to the enumeration order) and no shrinking is necessary. Another benefit is that we can systematically cover a bounded subset of the state space and can guarantee the absence of bugs for this subset. We implemented two exploration strategies:

```
def walkTree[T](root: Tree[T], breadth: Int = 2): LazyList[T] = {
  def walk(t: Tree[T], n: Int): LazyList[T] = {
    if (n <= 0) LazyList()
    else
      t.elem #:: (
        for {
          (c, i) <- t.children.take(n).zipWithIndex
          next <- walk(c, n - i / breadth - 1)
        } yield next)
  }

  LazyList.from(1).flatMap { depth =>
    walk(root, depth)
  }
}
```

Figure 7.5.: *Generic tree walking algorithm.*

Tree-Walking Strategy: In this first strategy, we view the possible executions as an infinite tree. Each state has a list of possible actions, which each leads to a successor state. To explore this tree, we use the algorithm shown in Figure 7.5. The function `walkTree` takes the root of the tree and a parameter `breadth`, which controls the ratio of exploring the breadth of the tree or the depth of the tree first. The result of the function is a lazy list of tree elements.

The nested `walk` function takes a tree node `t` and a depth `n`. If the depth reaches zero, the tree exploration stops. Otherwise, we return a stream consisting of the root element followed by elements from recursive calls for the children. We only take the first `n` children and invoke `walk` on the children with a reduced parameter for the depth. For the overall algorithm, we start from depth 1 and iteratively call `walk` with increasing values.

We stop the tree walking once an invariant violation is found or a timeout is reached. To reduce the size of the tree, we limit interleaving between different procedure invocations. We first finish a procedure invocation, before considering actions on another invocation.

Deduplication Strategy: This strategy keeps track of already visited states to avoid duplicate checks. This is similar to the basic idea behind the TLC model checker for TLA+ [YML99]. To detect equivalent states, we use a hashmap. States are checked for equivalence modulo renaming. The types that are considered for renaming are `InvocationId`, `TransactionId`, `CallId`, and all user defined types that are not algebraic data types. During equality checks, we build bijections between these values from the two states being compared. When computing hashes, we only count the number of occurrences of each value subject to renaming. Thus, the invocations `F(x,y)` and `F(z,x)` would result in the same hash value since they both have occurrences $(1, 1)$, while `F(z,z)` would result in a different hash value with occurrences (2).

The strategy then uses a work-list algorithm to explore the state space in a breadth-first manner. We restrict the number of values and invocations

```
case class SymbolicState(
  calls: SymbolicMap[SortCallId, SortCall],
  happensBefore: SymbolicMap[SortCallId, SortSet[SortCallId]],
  callOrigin: SymbolicMap[SortCallId, SortOption[SortTxId]],
  transactionOrigin: SymbolicMap[SortTxId, SortOption[SortInvocationId]],
  generatedIds: Map[IdType, SymbolicMap[SortCustomUninterpreted,
                                        SortOption[SortInvocationId]]],
  knownIds: Map[IdType, SymbolicSet[SortCustomUninterpreted]],
  invocationCalls: SymbolicMap[SortInvocationId, SortSet[SortCallId]],
  invocationOp: SymbolicMap[SortInvocationId, SortInvocationInfo],
  invocationRes: SymbolicMap[SortInvocationId, SortInvocationRes],
  currentInvocation: SVal[SortInvocationId],
  currentTransaction: Option[SVal[SortTxId]],
  localState: Map[ProgramVariable, SVal[_ <: SymbolicSort]],
  visibleCalls: SymbolicSet[SortCallId],
  currentCallIds: List[SVal[SortCallId]],
  trace: Trace[SymbolicState],
  internalPathConditions: List[NamedConstraint],
  snapshotAddition: SymbolicSet[SortCallId],
  translations: List[Translation]
)
```

Figure 7.6.: *Symbolic state structure used in Repliss.*

checked and iteratively widen these bounds once the state bounded state space is fully explored.

7.4. Symbolic Execution

We use symbolic execution to verify that a program is correct. The implementation of Repliss follows the proof rules for symbolic execution we introduced in Section 6.2, where we also described why we chose symbolic execution over alternatives like weakest precondition calculations.

7.4.1. Symbolic State

The structure of symbolic states is shown in Figure 7.6. We use the type `SVal[T]` for symbolic values with type `T`. The Type `SymbolicMap[K, V]` is short for `SVal[SortMap[K, V]]` and represents a total map[3] with keys of type `K` and values of type `V`. The Type `SymbolicSet[T]` is short for `SVal[SortSet[T]]` and represents sets of elements of type `T`. The types `Map`, `List`, and `Option` are the normal Scala types and therefore are not symbolic.

The symbolic state has the same structure as described in Figure 6.10 on page 99. However, there are some small adaptations in the data structures used to make it easier to express updates with the native functions provided by SMT solvers:

[3] In the context of SMT solvers, maps are also called *arrays*.

1. In Isabelle, we used a partial map (a map with values of type `option`) for the `calls` field. To avoid the indirection of `option`, we extended the type for calls (`SortCall`) with a constructor `NoCall`, which represents non-existent calls. We used the same technique for `invocationOp` and `invocationRes`.

2. The happens-before relation in Isabelle was represented as a relation (a set of pairs). In the Repliss tool, we use a map, which maps a database call to the set of database calls that happened before it. This makes it very easy to express updates for new database calls with the operations available in SMT solvers.

3. For the known and generated unique identifiers, we separate the state for the different user different id-types. This separation simplifies the generated uniqueness formulas.

4. We include the additional field `invocationCalls`, which stores the set of all database calls for each procedure invocation. Thus, it is the reverse of combining `callOrigin` and `transactionOrigin`. Again, this simplifies some formulas as it avoids quantification over the intermediate transaction layer.

Besides the actual symbolic state, the fields in Figure 7.6 also include the path conditions used for symbolic execution. Each path condition is represented by a `NamedConstraint`, which consists of a description, a priority, and a Boolean formula. The description is used when exporting the constraints to Isabelle and for showing users an overview of which constraints were actually used in the verification process. The weight of constraints is used to guide the SMT solver. Giving too many constraints the proof search might take too long, so we start with the constraints most likely to be used and then add further constraints incrementally.

We also keep a trace of the previous symbolic states, which we use to let a user go through a failing counter example step by step. Likewise, the field `translations` is also used for debugging and contains the verification condition exported to Isabelle and SMT-lib format.

The field `snapshotAddition` is used for an optimization we explain in Section 7.4.3

7.4.2. Implementing Rules

The implementation of the proof rules in symbolic execution is straight forward. The rules in 6.2 generally have the following form:

assumes $a_1 \dots a_n$
  and *cont*: $\bigwedge v_1 \dots v_m \,.\, p_1 \dots p_k \implies P\ S'\ result$
shows *execution-s-check Inv crdtSpec S stmt P*

We implement the rules in a function named `executeStatement` with the following signature:

```
def executeStatement(stmt: InStatement, state: SymbolicState,
              ctxt: SymbolicContext,
              follow: SymbolicState => SymbolicState): SymbolicState
```

The predicate P in the proof rules is the check executed after the command. In the implementation, we instead use the continuation parameter `follow`. This is a function that takes a state after the execution of `stmt` and executes the remainder of the code being checked, returning the final state.

A proof rule can have assumptions $a_1 \dots a_n$. These are conditions that must be shown to hold true. For example at the end of a transaction, we must show that the invariant still holds. In the implementation, this is done by invoking the SMT solver to check that these conditions are implied by the current path conditions. Technically, this is done by asking the SMT solver whether the conjunction of all path conditions with the negation of an a_i is satisfiable. If the SMT solver returns that it is unsatisfiable, we have verified the necessary implication. In the case it returns a model satisfying the conjunction, we have a counter example that we can present to the user. If it returns *unknown*, the verification was not successful, and we can only present a partial counter example.

After the necessary checks are performed, we consider the *cont* premise in our proof rule, which represents the continuation. This can introduce new symbolic variables $v_1 \dots v_m$, new path conditions $p_1 \dots p_k$, and a new symbolic state S'. In the implementation this is reflected by adding the path conditions to the state and then calling the continuation function (`follow`) with the new state S'.

Execution of Branching Statements In Section 6.2, we have not introduced any proof rule for branching statements like if-statements. This was not necessary, since the shallow embedding allowed us to use Isabelle's own conditional expression and the corresponding rules for splitting the subgoal. In the implementation of Repliss, we use an equivalent strategy and split the execution into two execution paths. We invoke the SMT solver to detect infeasible branches so that we avoid executing the complete path to the end of the procedure in such a case. Nevertheless, this splitting rule leads to a number of paths exponential in the number of branching constructs taken. In principle, it would be possible to rejoin two states after a branching construct to avoid the exponential blowup. This strategy would result in fewer calls to the SMT solver, albeit with more complex formulas. Since SMT solvers can handle branching more efficiently, this can give some performance improvements as shown in experiments for another symbolic execution tool [KMS12]. However, we have not implemented this optimization in Repliss, as joining two symbolic states would require some implementation effort. As the symbolic state is optimized for a single execution path, joining requires more work than merely calculating a disjunction of two formulas.

7.4.3. Adaptations

The implementation contains some adaptations compared to a straight forward implementation of the proof rules. These are techniques to simplify the job of the SMT solver and to exploit the type information that we have in Repliss programs but not in the Isabelle formalization.

Prioritized Constraints. During symbolic execution many path conditions are collected. In particular, the predicates *wellformed* and *state-monotonic-Growth* are implemented using a conjunct of many constraints. Sending all the collected constraints to the SMT solver at once can overwhelm it, so that it can neither find a proof nor a counter example. To address this issue, we assign priorities to the generated assertions. A higher priority is assigned if the invariant is more likely to be used in a proof. Lower priorities are used for constraints that are hard to use for an SMT solver because of their structure. We identified these constraints experimentally by measuring which constraints had a significant impact on the time to find a proof or a counter example.

Given the set of prioritized constraints, we invoke the SMT solver incrementally. We start with a selection of the highest priority constraints. When the constraints are found to be unsatisfiable we are done, since then the overall set of constraints is unsatisfiable as well. When we find a counter-example, we add the constraints with the next lower priority level and again let the SMT solver check the constraints. If there are no more constraints to add, verification has failed and we have a counter example to present to the user. If the solver returns *unknown*, we assume that this result would not change by adding more constraints, since the solver apparently was already overwhelmed. In case we previously found a counter example with fewer constraints, we return this counter example but flag it as incomplete.

Handling Quantification over all Valid Snapshots. As we described in Section 7.1, database queries can only be used in the context of a database snapshot. For specifications, we have a special construct to quantify over all valid snapshots. Unfortunately, this quantification is not easy to instantiate for an SMT solver. We address this issue by instantiating the quantifier specifically for two special cases. We only optimize for the case, where the quantifier appears at the outer level. In that case, we will typically have one path constraint from assuming the invariant at the beginning of a transaction, where the quantifier appears positively and another appearance from the proof obligation at the end of the transaction, where it is negated. The negated quantifier can be seen as an existential quantification, for which we can create a fresh symbolic value representing the snapshot. Then we can consider two cases for this snapshot: Either it includes the transaction that was just committed or it does not. The first case is only feasible, if the snapshot is a superset or is equal to the snapshot of the transaction being committed, as otherwise causal consistency would be violated.

In both cases, we can extract the part of the snapshot that does not include

the new transactions, which we store as `snapshotAddition` in the symbolic state. The quantifier stemming from the old invariant at the beginning of the transaction is automatically instantiated with the `snapshotAddition` set. Thus, if the new transaction is included, the SMT solver has to check that the invariant for a snapshot is preserved when adding a single transaction to the snapshot. In the other case, it merely has to prove that it is preserved for the same snapshot, which is trivial in many cases.

Using Types. In contrast to our Isabelle theories, the Repliss tool uses a typed language. This can be exploited in symbolic execution. First, it simplifies the translation to SMT solvers, which also uses a typed input language. In Isabelle, we needed to use a union type to cover the different variants for values and accordingly we had to use conversion functions and case distinctions which complicates formulas.

With using types, we can also make the tracking of unique identifiers more precise. Instead of a single data structure for all unique identifiers, we can partition this structure by type. This can be seen in the fields `generatedIds` and `knownIds` in the symbolic state (Figure 7.6 on page 138).

7.4.4. Design Decisions

In the first version of Repliss, we used Boogie [Bar+05] as an intermediate language for verification. Boogie is a language designed to be an intermediate language for verification of sequential, imperative programs. Specifications are given by pre- and post-conditions of procedures and assertions. The Boogie tool calculates weakest preconditions which are then checked by the Z3 [**Z3**] theorem prover. We encoded the rules of the single-invocation semantics in pre- and post-conditions of procedures.

While this approach worked with some initial examples, it was hard to debug cases, where Z3 was not able to prove the generated verification conditions. We therefore changed from Boogie to Why3 [FP13]. This required little changes to the code, since both languages provide the same core features. Targeting Why3 is slightly more complex, since the language has more syntactic restrictions and not primarily designed as an intermediate language. However, one big advantage of Why3 is its IDE, which allows trying different tactics and provers for the generated verification conditions. It also allows splitting verification conditions and run solvers on smaller parts. The parts that cannot be handled by automated theorem provers can then be exported to interactive provers like Isabelle, which is helpful for debugging problems in verification attempts. However, the export does not generate code with stable and readable identifiers. It simply numbers all the assumptions, which means that a small change in the generated code can result in many necessary changes in the interactive proof.

Another aspect is the generation of counter examples for failed proof attempts. While both Boogie and Why3 support the generation of counter examples in principle, in our examples they were never able to produce exam-

ples. When we switched to our own implementation, we had more control over counter example generation. First, we could invoke CVC4 with the special option for finding finite models [Rey+13], which turned out to work well with the formulas we generate. Moreover, with more control over the interaction with SMT solvers, we can pass assumptions to the solver incrementally. It turned out, that some assumptions about the well-formedness of states make it very hard for the solvers to find counter examples. When these assumptions are added later, we can at least get a preliminary counter example that does not respect all constraints but still can give a hint on potentially missing invariants for the program to be verified.

The additional control over generated formulas also allows us to generate more efficient formulas. For example, we can collect all fresh call identifiers on an execution path and generate a formula that they are distinct, for which solvers provide a builtin predicate. Moreover, we can generate meaningful names for added constraints, which are unlikely to be changed when minor aspects of a program or the Repliss implementation are changed.

7.5. Predicate Abstraction for Verification

The proof rules we derived in Section 6.2 make use of the two predicates *state-monotonicGrowth* and *wellformed*. The problem with these is that they are defined semantically (based on the interleaving semantics), which makes them unsuitable for a direct translation to SMT solvers. Instead, our approach is to over-approximate these predicates. To this end, we use properties that we proved in Section 5.3.1 to characterize the predicates. This is an over-approximation in the sense that other predicates might also satisfy the same predicates.

The challenge here is to include all predicates which might be required for verifying applications. However, each predicate also increases the input for the SMT solver and thus can slow down the verification process.

7.5.1. Well-formed States

We first consider the predicate *wellformed*, which we defined formally (see page 121) as any state reachable from the initial state. We have used Isabelle to verify several properties that hold true for all *wellformed* states. In the following we present all of these lemmas that we use in the Repliss to characterize *wellformed* states. Of course, this is not a complete representation of the property. We made the selection based on counter examples found when working with examples in Repliss. Whenever Repliss produced counter examples that violated expected consistency guarantees, we added corresponding constraints to the tool.
The happens-before relation only is defined on valid database calls, which is the domain of the *call* map:

lemma *happensBefore-in-calls-left*:
assumes *wf*: *state-wellFormed S*

```
    and (x,y)∈happensBefore S
shows x∈dom (calls S)

lemma happensBefore-in-calls-right:
assumes wf: state-wellFormed S
    and (x,y)∈happensBefore S
shows y∈dom (calls S)
```

The visible calls are a subset of all valid database calls.

```
lemma state-wellFormed-vis-subset-calls:
  assumes state-wellFormed S
    and visibleCalls S i ≜ vis
    and c ∈ vis
shows c ∈ dom (calls S)
```

The call origin is defined for all calls:

```
lemma wellFormed-callOrigin-dom3:
  assumes a1: state-wellFormed S
  shows (calls S c = None) ⟷ (callOrigin S c = None)
```

In the same invocation, all calls are totally ordered:

```
lemma state-wellFormed-same-invocation-sequential:
  assumes state-wellFormed S
    and callOrigin S c1 ≜ tx1
    and txOrigin S tx1 ≜ i
    and callOrigin S c2 ≜ tx2
    and txOrigin S tx2 ≜ i
    and c1 ≠ c2
shows (c1,c2)∈happensBefore S ∨ (c2,c1)∈happensBefore S
```

The current snapshot is transactionally and causally consistent (see Section 5.3.1):

```
lemma wf-transactionConsistent-noTx:
  assumes wf: state-wellFormed S
    and visibleCalls S i ≜ vis
    and currentTx S i = None
  shows transactionConsistent (callOrigin S) (txStatus S) vis

lemma wf-causallyConsistent1:
  assumes wf: state-wellFormed S
    and visibleCalls S i ≜ vis
  shows causallyConsistent (happensBefore S) vis
```

The happens-before relation is irreflexive, antisymmetric, and transitive:

```
lemma happensBefore-irrefl:
assumes wf: state-wellFormed S
shows irrefl (happensBefore S)
```

lemma *state-wellFormed-hb-antisym*:
assumes *state-wellFormed S*
assumes $(x,y) \in$ *happensBefore S*
shows $(y,x) \notin$ *happensBefore S*

lemma *happensBefore-transitive*:
assumes *wf*: *state-wellFormed S*
shows *trans* (*happensBefore S*)

There can be no procedure invocation result without a procedure invocation:

lemma *state-wellFormed-invocation-before-result*:
assumes *state-wellFormed C*
and *invocOp C s* = *None*
shows *invocRes C s* = *None*

The happens-before relation is consistent for calls from the same transaction.

lemma *wf-transaction-consistent-l*:
assumes *state-wellFormed S*
and *callOrigin S y1* = *callOrigin S y2*
and *callOrigin S x* ≠ *callOrigin S y1*
and (*y1*, *x*)∈*happensBefore S*
shows (*y2*, *x*)∈*happensBefore S*

lemma *wf-transaction-consistent-r*:
assumes *state-wellFormed S*
and *callOrigin S y1* = *callOrigin S y2*
and *callOrigin S x* ≠ *callOrigin S y1*
and (*x*, *y1*)∈*happensBefore S*
shows (*x*, *y2*)∈*happensBefore S*

If a transaction has no originating invocation, there can be calls in the transaction:

lemma *state-wellFormed-transactionOrigin-callOrigin*:
assumes *state-wellFormed S*
and *txOrigin S tx* = *None*
shows *callOrigin S c* ≠ *Some tx*

If a procedure invocation is not defined, there can be no transactions in it:

lemma *wf-no-invocation-no-origin*:
assumes *state-wellFormed S*
and *invocOp S i* = *None*
shows *txOrigin S tx* ≠ *Some i*

Besides the properties above, the *wellformed* property in the Repliss tool also includes the constraints about unique identifiers, which we presented in Section 6.3. We also include definitions and constraints for the auxiliary field that are not present in the Isabelle theories. These are the `snapshotAddition` field and the `invocationCalls` field.

7.5.2. State Monotonic Growth

We now consider the predicate *state-monotonicGrowth*, which relates two states S and S' under a current invocation i. The formal definition of *state-monotonicGrowth*(i, S, S') (see page 62) states that the new state S' is reachable from the old state S with steps in invocations different from i. To characterize this property without referring to the semantics, we use the properties below, which capture the monotonicity of the property.

When a call exists in the old state, is has the same information in the new state:

lemma *state-monotonicGrowth-calls*:
assumes *state-monotonicGrowth i S S'*
shows *calls S c ≜ info ⟹ calls S' c ≜ info*

lemma *state-monotonicGrowth-callOrigin*:
assumes *state-monotonicGrowth i S S'*
and *callOrigin S c ≜ t*
shows *callOrigin S' c ≜ t*

lemma *state-monotonicGrowth-callOrigin-unchanged*:
assumes *state-monotonicGrowth i S S'*
and *calls S c ≠ None*
shows *callOrigin S c ≜ tx ⟷ callOrigin S' c ≜ tx*

lemma *state-monotonicGrowth-happensBefore*:
assumes *state-monotonicGrowth i S S'*
shows *c2∈dom (calls S) ⟹ ((c1,c2)∈happensBefore S' ⟷ (c1,c2)∈happensBefore S)*

Similarly, when a transaction exists in the old state, it is unchanged in the new state:

lemma *state-monotonicGrowth-transactionOrigin*:
assumes *state-monotonicGrowth i S S'*
and *txOrigin S t ≠ None*
shows *txOrigin S t ≜ i' ⟷ txOrigin S' t ≜ i'*

Moreover, no new database calls can be added to a committed transaction:

lemma *state-monotonicGrowth-no-new-calls-in-committed-transactions*:
assumes *state-monotonicGrowth i S S'*
and *callOrigin S' c ≜ tx*
and *calls S c = None*
shows *txStatus S tx ≠ Some Committed*

Monotonicity also holds for the invocation history.

lemma *state-monotonicGrowth-invocOp-unchanged*:
assumes *state-monotonicGrowth i S S'*
and *invocOp S i' ≠ None*
shows *invocOp S i' = invocOp S' i'*

lemma *state-monotonicGrowth-invocRes-unchanged*:
assumes *state-monotonicGrowth i S S'*
and *invocRes S i' ≠ None*
shows *invocRes S i' = invocRes S' i'*

```
invariant shape_of_invocation_editMessage:
  forall invoc: InvocationId, param_id: MessageId, param_newContent: String ::
    invoc.info == editMessage(param_id, param_newContent))
      ==>  (invoc.result == NoResult()
             && (forall tx: TransactionId :: !(tx.origin == invoc)))
        || (exists tx_0: TransactionId ::
              (forall tx: TransactionId ::
                    tx.origin == invoc ==> tx_0 == tx)
             && (exists c_0: CallId :: forall c: CallId ::
                    c.tx == tx_0 ==> c == c_0)
             && c_0.tx == tx_0
             && c_0.op == Qry(messageQry(ContainsKey(param_id))))
        || (exists tx_0: TransactionId ::
               (forall tx: TransactionId ::
                  tx.origin == invoc ==> tx_0 == tx)
              && (exists c_0: CallId, c_1: CallId ::
                       distinct(c_0, c_1)
                    && (forall c: CallId ::
                        c.tx == tx_0 ==> c == c_0 || c == c_1)
                    && c_0.tx == tx_0
                    && c_1.tx == tx_0
                    && c_0.op == Qry(messageQry(ContainsKey(param_id)))
                    && c_1.op == Op(message(NestedOp(param_id,
                                        content(Assign(param_newContent)))))
                    && c_0 happensBefore c_1))

invariant shape_rev_Op_message_NestedOp_MessageId_author_Assign:
    forall c: CallId, x: MessageId, y: UserId ::
        c.op == Op(message(NestedOp(x, author(Assign(y)))))
        ==> (exists invoc: InvocationId, text: String ::
                c.origin == invoc
             && invoc.info == sendMessage(y, text))
```

Figure 7.7.: *Shape invariant for* `editMessage` *procedure and reverse shape invariant for assignments to author field.*

7.6. Shape Invariants

Some invariants can easily be derived from the application source code, so we do not require them to be written down explicitly. Similar to shape analysis [WSR00] for finding heap invariants in heap-manipulating problems, we can derive shape invariants for the relation between procedure invocations and the corresponding database history.

We automate the generation of invariants only for loop-free procedures by abstractly executing each path through the procedure. From this we generate an invariant that enumerates all possible sequences of transactions and database calls which can occur in the invocation. Correspondingly, we also generate invariants for the inverse direction. For each kind of database operation, we generate an invariant listing all the possible kinds of invocations that could trigger the database call. Figure 7.7 shows both kinds of shape invariants for the invocations of `editMessage` and database calls to `message_content_assign`.

The abstract values we use during the abstract interpretation of a path are

the following:

```
AbstractValue =
      BoolValue(value: Boolean)
    | DatatypeValue(typ: InTypeExpr, constructorName: String,
                    args: List[AbstractValue])
    | ParamValue(paramName: String, typ: InTypeExpr)
    | AnyValue(name: String, typ: InTypeExpr)
```

At the start of each path, the procedure arguments are assigned to corresponding `ParamValue` values. Expressions are evaluated abstractly and if a value cannot be represented by any of the first three cases, it is handled as an `AnyValue`. While interpreting the execution path, we maintain a list of executed transactions with a list of database calls in the transactions. For each database call, we store the operation, which typically is a `DatatypeValue`.

With the results from the analysis, we can create an invariant for the shape of each procedure. A possible state for a procedure is a prefix of the transactions in one of the procedure's execution paths. For the invariant, we create a disjunction of all these possible states and describe the transactions with the contained database calls based on the information we have from the analysis. Abstract values of type `ParamValue` can be related to the procedure invocation information. Abstract values of type `AnyValue` are handled by introducing existentially quantified variables. If the prefix is not a complete path, we furthermore add the constraint that the procedure does not have a returned result yet.

For generating the reverse invariants (from database calls to originating invocations), we can use the results from the same analysis[4]. To this end, we enumerate all possible database operations with placeholders for the variables. For non-recursive datatypes, we also enumerate the possible variants of the datatype. For example Figure 7.8 shows the enumeration of operations for the chat example with the different levels of nesting. Variables are denoted with a hash sign (`#`).

We then go through the results of the shape analysis and match the operations in the analysis with the enumerated operation patterns, unifying variables in the process. This gives us a list of possible procedure invocations for each database operation. From this we can create an invariant by creating a disjunction of the different cases. When a variable in an operation matches with a parameter value, we express this relation in the invariant. Otherwise, we again introduce existentially quantified variables.

[4]The reverse direction only uses parts of the information gathered in the analysis. It would be possible to do a simpler analysis for this direction, that does not require checking each execution path individually. This analysis could then also work with loops using the normal abstract execution framework with `AnyValue` being the bottom value of the lattice.

```
Op(#x_1)
Op(chat(#x_2))
Op(chat(Add(#x_3)))
Op(chat(Remove(#x_4)))
Op(message(#x_5))
Op(message(DeleteKey(#x_6)))
Op(message(NestedOp(#x_8, #x_7)))
Op(message(NestedOp(#x_8, author(#x_9))))
Op(message(NestedOp(#x_8, author(Assign(#x_10)))))
Op(message(NestedOp(#x_8, content(#x_11))))
Op(message(NestedOp(#x_8, content(Assign(#x_12)))))
```

Figure 7.8.: *Enumeration of operations in the Chat example.*

Chapter 8

Evaluation

For the evaluation of the tool, we have implemented a few applications in Repliss and using our Isabelle framework. This chapter begins with descriptions of the case studies and their realization in our Isabelle framework. In Section 8.4, we then evaluate the performance of the automated testing and verification in Repliss.

8.1. Chat Example

In Section 2.1 we introduced our running example of the Chat application and showed how it can be verified in the Repliss tool. We now show how our formalized proof technique can be instantiated within Isabelle/HOL to prove its correctness.

8.1.1. Modelling the application

As the first step we need to model the application in Isabelle. This requires us to fix a type for the values to be used in the program. To this end we define the type *val* in Figure 8.1. This sum type includes general purpose values such as *String*, *Bool*, or the value *Undef*, but it also includes application specific datatypes like users, chats, and messages. We also include cases for the possible return values of the *getMessage* procedure, which can return *NotFound* or *Found* with the respective data.

Next, we define the replicated data structures used for the persistent storage. For this we can use the operations and specifications defined in Section 4.1. We only need to define custom datatypes for the structs we want to use in the data-model. For the chat example these are *messageDataOp* and *operation* datatypes defined in Figure 8.2.

The *messageDataOp* type is for storing data of a message and is a struct with two fields, for the author and the content of the message. Both fields are registers with initial value *Undef*, which we define in the definition of *messageStruct* in Figure 8.2.

```
datatype val =
    String string
  | Bool bool
  | Undef
  | UserId int
  | ChatId int
  | MessageId int
  | Found val val
  | NotFound
```

Figure 8.1.: *Value type for the chat application.*

```
datatype messageDataOp =
    Author (val registerOp)
  | Content (val registerOp)

datatype operation =
    Chat (val setOp)
  | Message ((val, messageDataOp) mapOp)

messageStruct ≡
struct-field Author (register-spec Undef) .∨.
struct-field Content (register-spec Undef)

crdtSpec ≡
struct-field Message (map-sdw-spec messageStruct) .∨.
struct-field Chat set-rw-spec
```

Figure 8.2.: *Replicated datatype specification for the chat application.*

The *operation* type is for toplevel operations, for which we give the specification in *crdtSpec*. The struct consists of two fields: *Message* is a map with *sdw* strategy and using the *messageStruct* for the values in the map. *Chat* is a set with *rw* strategy, storing the set of all messages in the chat.

Having defined the structure and semantics for the persistent data, in the next step we present the implementation of the procedures, which we give in Figure 8.3. We use Isabelle's monad-syntax (do-notation) for the implementation, which lets us write the code very similar to the code we presented earlier in Figure 2.2 on page 7. The biggest transformation is caused by the fact that we cannot have side-effects in expressions. Therefore, the database queries that we previously used in an expression have to be moved to their own statements and their results have to be bound to new variables. We also need to consider that every value in the program is of type *val*, so instead of simply writing *if exists then* in an if-statement, we have to use a comparison.

Finally, the datatype *proc* in Figure 8.4 defines the available procedures. Note that the design of this type makes it impossible to invoke a procedure with arguments of a wrong type. We only convert the values to the dynamic type *val* in the function *procedures*, which is given below the datatype. Based

```
definition sendMessage-impl :: val ⇒ val ⇒ (val,operation,val) io where
  sendMessage-impl from content ≡  do {
  m ← newId isMessageId;
  atomic (do {
    call (Message (NestedOp m (Author (Assign from))));
    call (Message (NestedOp m (Content (Assign content))));
    call (Chat (Add m))
  });
  return m
}

definition editMessage-impl :: val ⇒ val ⇒ (val,operation,val) io where
  editMessage-impl m newContent ≡  do {
  atomic (do {
    exists ← call (Message (KeyExists m));
    if exists = Bool True then
      call (Message (NestedOp m (Content (Assign newContent))))
    else
      skip
  })
}

definition deleteMessage-impl :: val ⇒ (val,operation,val) io where
  deleteMessage-impl m ≡  do {
  atomic (do {
    call (Chat (Remove m));
    call (Message (DeleteKey m))
  })
}

definition getMessage-impl :: val ⇒ (val,operation,val) io where
  getMessage-impl m ≡  do {
  atomic (do {
    exists ← call (Message (KeyExists m));
    if exists = Bool True then do {
      author ← call (Message (NestedOp m (Author Read)));
      content ← call (Message (NestedOp m (Content Read)));
      return (Found author content)
    } else do {
      return NotFound
    }
  })
}
```

Figure 8.3.: *Procedure implementations for the chat application.*

```
datatype proc =
    SendMessage string string
  | EditMessage int string
  | DeleteMessage int
  | GetMessage int

definition procedures :: proc ⇒ (localState × (localState, operation, val) procedureImpl) where
  procedures invoc ≡
  case invoc of
    SendMessage author content ⇒ toImpl′ invoc (sendMessage-impl (String author) (String content))
  | EditMessage m newContent ⇒ toImpl′ invoc (editMessage-impl (MessageId m) (String newContent))
  | DeleteMessage m ⇒ toImpl′ invoc (deleteMessage-impl (MessageId m))
  | GetMessage m ⇒ toImpl′ invoc (getMessage-impl (MessageId m))
```

Figure 8.4.: *Procedure dispatch function for the chat application.*

on a given *proc*, this function chooses the corresponding implementation.

8.1.2. Proving application correctness

After modelling the application, we can now verify its correctness. As in Section 2.1 we are going to verify the property that whenever *GetMessage* returns *Found* with some author, then there must be a corresponding procedure invocation of *SendMessage* with the same author. This property is formalized as *inv1* in Figure 8.5.

Note that the parameters of *inv1* do not contain the complete invariant context, but only information about procedure operations and results. Only the combination of all invariants (defintion *inv* in Figure 8.5) takes the whole context and then passes the relevant parts to each invariant. This makes it easier to later handle the case where the parameters of an invariant do not change.

Besides our specification *inv1*, we need three additional invariants for the verification to succeed. Invariants *inv2* and *inv3* are the similar to the ones presented in 2.1.4. Invariant *inv4* is a shape invariant that is automatically inferred by the Repliss tool (see Section 7.6) and thus did not appear explicitly in Section 2.1.4. We quickly recap the essence of the auxiliary invariants from Figure 8.5:

inv2 If there is an assignment to the *Content* field, there also must be an assignment to the *Author* field that happened before.

inv3 There is no update on a message after it has been deleted.

inv4 If there is an assignment to the author field, then there is a procedure invocation of *SendMessage* with the same author.

$$
\begin{aligned}
&\mathit{inv1}\ \mathit{op}\ \mathit{res} \equiv \\
&\forall\ g\ m\ \mathit{author}\ \mathit{content}. \\
&\quad \mathit{op}\ g \triangleq \mathit{GetMessage}\ m \wedge \mathit{res}\ g \triangleq \mathit{Found}\ (\mathit{String}\ \mathit{author})\ \mathit{content} \longrightarrow \\
&\quad (\exists\ s\ \mathit{content2}.\ \mathit{op}\ s \triangleq \mathit{SendMessage}\ \mathit{author}\ \mathit{content2})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inv2}\ \mathit{cop}\ \mathit{hb} \equiv \\
&\forall\ \mathit{c1}\ m\ s. \\
&\quad \mathit{cop}\ \mathit{c1} \triangleq \mathit{Message}\ (\mathit{NestedOp}\ m\ (\mathit{Content}\ (\mathit{Assign}\ s))) \longrightarrow \\
&\quad (\exists\ \mathit{c2}\ u. \\
&\qquad \mathit{cop}\ \mathit{c2} \triangleq \mathit{Message}\ (\mathit{NestedOp}\ m\ (\mathit{Author}\ (\mathit{Assign}\ u))) \wedge (\mathit{c2},\ \mathit{c1}) \in \mathit{hb})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inv3}\ \mathit{cop}\ \mathit{hb} \equiv \\
&\nexists\ \mathit{write}\ \mathit{delete}\ m\ \mathit{no}. \\
&\quad \mathit{cop}\ \mathit{write} \triangleq \mathit{Message}\ (\mathit{NestedOp}\ m\ \mathit{no}) \wedge \\
&\quad \mathit{is\text{-}update}\ \mathit{no} \wedge \mathit{cop}\ \mathit{delete} \triangleq \mathit{Message}\ (\mathit{DeleteKey}\ m) \wedge (\mathit{delete},\ \mathit{write}) \in \mathit{hb}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inv4}\ \mathit{op}\ \mathit{cop} \equiv \\
&\forall\ c\ m\ u. \\
&\quad \mathit{cop}\ c \triangleq \mathit{Message}\ (\mathit{NestedOp}\ m\ (\mathit{Author}\ (\mathit{Assign}\ (\mathit{String}\ u)))) \longrightarrow \\
&\quad (\exists\ i\ s.\ \mathit{op}\ i \triangleq \mathit{SendMessage}\ u\ s)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inv}\ \mathit{ctxt} \equiv \\
&\mathit{inv1}\ (\mathit{invocOp}\ \mathit{ctxt})\ (\mathit{invocRes}\ \mathit{ctxt}) \wedge \\
&\mathit{inv2}\ (\mathit{Op}\ \mathit{ctxt})\ (\mathit{happensBefore}\ \mathit{ctxt}) \wedge \\
&\mathit{inv3}\ (\mathit{Op}\ \mathit{ctxt})\ (\mathit{happensBefore}\ \mathit{ctxt}) \wedge \mathit{inv4}\ (\mathit{invocOp}\ \mathit{ctxt})\ (\mathit{Op}\ \mathit{ctxt})
\end{aligned}
$$

Figure 8.5.: *Invariants for the chat application.*

After defining the invariants, we need to show that they are valid for the chat application. The proof that the invariants hold in the initial state is trivial, since all invariants quantify over the history and initially there are neither database calls nor any procedure calls in the history. So it remains to be shown that each procedure invocation maintains the invariants.

Procedure SendMessage

First, we need to show that the invariant is maintained right after the invocation has started. Here the only state change is that the *SendMessage* invocation is added to the history of procedure invocations. This change can only affect invariants 1 and 4 as the others do not depend on this information. In both cases, the newly added invocation of *SendMessage* cannot invalidate the invariant, if we consider the fact that the invocation is fresh and thus cannot overwrite any previous invocation in the history.

Next, we show that executing the body of *SendMessage* maintains the invariant. Here, we use our tactics for symbolic execution (see Section 6.2.4) which yields two cases where we need to show that the invariant is maintained: when committing the transaction and when returning from the procedure.

At Commit. As transactions affect the database history, we have to consider invariants 2, 3, and 4 when committing the transaction.

Invariant 2 states for every assignment to the *Content* field, there is an assignment to the *Author*. This is trivial for the new transaction, since it contains both calls in the desired order. It is also clear that the invariant is maintained for the old part of the history and thus thus invariant 2 can be discharged automatically.

Invariant 3 states that there is no update on a message after is has been deleted. This property is a bit more involved, since we are performing updates on a message in the transaction. So we need to show that these updates are not on a deleted message, which cannot be the case since we are working on a fresh message identifier. To reason about unique identifiers, we use the property *uid-is-private'* which we get from the rule *execution-s-check-beginAtomic* (see Figure 6.20) at the start of the transaction. This property implies that no database call can include the just created message identifier. In particular there can be no delete operation on it. Using this fact, it is easy to show that invariant 3 is maintained.

Finally, invariant 4 states that each assignment of the *Author* field has a corresponding invocation of *SendMessage*. In our new transaction, we obviously have both so this invariant is again easy to show.

At Return. When returning from the procedure, only the invocation result changes. This only affects invariant 1, for which we can automatically show that it is maintained.

Procedure EditMessage

Again, the invariant check right at the start of the invocation is trivial as no invariants refer to invocations of *EditMessage*.

When executing the procedure with our symbolic execution method, we end up with four proof obligations: We have to consider the two cases of the if-condition and for both cases we have to show that the invariant is maintained at transaction commit and when returning from the procedure. The proof obligation for returning from the procedure is trivial in both cases, since invocation results for *EditMessage* do not appear in the invariants. The case that the if-condition is false (the message does not exist) also is trivial, since no updates are performed in that case. This leaves us with one interesting case.

If the if-condition evaluated to
True, we can use the CRDT specification to obtain information about prior database calls. Since the entry for message m exists, there must be a database call *upd-c*, such that *upd-c* is an visible update on message m with an update operation *upd-op*. Also there is no delete operation on message m after *upd-c*. If we combine this with invariant 3, we get that there is no visible delete operation for message m.

With this information alone, we do not know whether *upd-op* is an update of the *Author* field or of the *Content* field. However, in the latter case we can instantiate invariant 2 to obtain a database call *upda-c*, which is an update to the *Author field* and which happened before *upd-c*. Using causal consistency, we get that *upda-c* is also visible.

Thus, we can always find an update operation of the author field, which is visible at the start of the transaction. This shows that invariant 2 is maintained for the new update on the content of message m in the transaction.

For invariant 3, we have to show that the update performed in the transaction does not occur after a delete operation. This is the case as there cannot be any delete operation for the message, as we derived earlier.

Invariants 1 and 4 are trivially maintained as they are not affected by the transaction.

Procedure DeleteMessage

This procedure is easily verified, since it only performs two unconditional updates. Since we are performing a delete-operation in the transaction, for invariant 3 we need to show that there cannot be any update after the delete. This is a general property in the proof rule for committing a transaction (see Figure 6.23 on page 114).

Procedure GetMessage

Since this procedure only performs queries, the only interesting case in this procedure is when the message exists and we return *Found*. Since we return

Found from *GetMessage*, for invariant 1 we need to show that there is a corresponding invocation of *SendMessage* with the same author.

If we use the query specification for reading the author field, we obtain an assignment operation with the value stored in the register. Then we can use invariant 4 to get a corresponding procedure invocation of *SendMessage* which we can use to prove invariant 1.

8.2. User Database

The user database example demonstrates how the effect of a procedure can be specified using the happens-before relation on invocations. We model a simple user database in which users can be registered and their account information can be retrieved. User accounts can also be updated and removed from the database.

The Repliss code for the example is given in Figure 8.6. We use a delete-wins map CRDT named `user` to store the user values. The key of the map is the user identifier and the value is a struct consisting of two string-registers storing the name and mail address of the user.

We want to verify that the `removeUser` procedure works correctly. This means that trying to read a user after it has been removed should yield the result `notFound`. This is expressed by the following invariant:

```
invariant (forall r: InvocationId, g: InvocationId, u: UserId  ::
    r.info == removeUser(u)
 && g.info == getUser(u)
 && r happened before g
 ==> g.result == getUser_res(notFound()))
```

This invariant alone is not sufficiently strong for proving itself. When we run Repliss on the example, we get a counter example, where `registerUser` is called after `removeUser`. Thus, there is an update after removing the user and a subsequent call to `getUser` would return `found`. However, in the actual system, this execution is not possible as `registerUser` always produces a new `UserId` and thus cannot result in an update after a remove. We therefore add another invariant, which states that there can be no update operation after a remove:

```
invariant !(exists write: CallId, delete: CallId,
    u: UserId, upd ::
       write.op == Op(user(NestedOp(u, upd)))
    && delete.op == Op(user(DeleteKey(u)))
    && delete happened before write)
```

When checking the `registerUser` procedure, this invariant is automatically verified by the fact that a new unique identifier is used. In the check for `getUser` it can then be used to show that after a remove, the `ContainsKey` query cannot return true.

Failing Variants In order to check the bug finding functionality of Repliss, we also implemented two variants of the userbase example that contain bugs.

```
def registerUser(uName: String, uMail: String): UserId
    var u: UserId
    atomic
        u = new UserId
        call user(NestedOp(u, name(Assign(uName))))
        call user(NestedOp(u, mail(Assign(uMail))))
    return u

def updateMail(id: UserId, newMail: String)
    atomic
        if userQry(ContainsKey(id))
            call user(NestedOp(id, mail(Assign(newMail))))

def removeUser(id: UserId)
    call user(DeleteKey(id))

def getUser(id: UserId): getUserResult
    atomic
        if userQry(ContainsKey(id))
            return found(userQry(NestedQuery(id, nameQry(
                ReadRegister))),
                         userQry(NestedQuery(id, mailQry(
                             ReadRegister))))
        else
            return notFound()

// used types:
idtype UserId
type String

type getUserResult =
    notFound()
  | found(name: String, mail: String)

// CRDT specifications
crdt user: Map_dw[UserId, {
    name: Register[String],
    mail: Register[String]
}]
```

Figure 8.6.: *Repliss Code for the user database example.*

1. In the first variant, we remove the atomic block from the `updateMail` procedure. This can lead to the case, where the `ContainsKey` check is performed before a user is removed, but the update afterwards. A subsequent call to `getUser` would then return `found` since there is an update after a delete.

2. In the second variant, we use an update-wins map CRDT to store the users. This leads to a bug, when the `updateMail` method is invoked concurrently to `removeUser`. Since the map has update-wins semantics, a subsequent call to `getUser` would then incorrectly return `found`.

8.2.1. Isabelle Formalization

Besides verifying the user database example with the Repliss tool, we also proved its correctness manually in our Isabelle framework. The implementation of the procedures is shown in Figure 8.7 and the invariants for the application in Figure 8.8. The procedure implementation is equivalent to the implementation in Repliss. For the invariants, we need *inv3* as an additional explicit invariant. In the Repliss tool, this invariant is covered by the automatically generated shape invariants.

We now go through the procedures and shortly describe the necessary steps for the verification.

Procedure RegisterUser

After Invocation. The invariants hold directly after the invocation, since *RegisterUser* does not appear in the invariants.

At Commit. Invariant 1 and 3 are not affected by the transaction. For invariant 2, we have to show that there can be no previous delete operation on the same user. To show this, we need to use the fact that we have created a new unique identifier, and with the *uid-is-private'* property, we get that there cannot be any database calls containing involving this user.

At Return. The procedure return does not affect any invariant, so this case is again trivial.

Procedure UpdateMail

Similar to the case for *RegisterUser*, only the proof obligation at transaction commit is interesting here.

When the if-condition evaluates to *true*, we perform an update, so for *inv2* we have to show that there can be no delete-operation before it. We can use the CRDT specification of the *KeyExists* query to show that there must be an update for which all delete operations happened before. Combining this with *inv2* from the pre-state, we get that there can be no visible delete operations. With this it is easy to show that the invariant is maintained.

```
definition registerUser-impl :: val ⇒ val ⇒ (val,operation,val) io where
  registerUser-impl name mail ≡  do {
  u ← newId isUserId;
  atomic (do {
    call (NestedOp u (Name (Assign name)));
    call (NestedOp u (Mail (Assign mail)))
  });
  return u
}

definition updateMail-impl :: val ⇒ val ⇒ (val,operation,val) io where
  updateMail-impl u mail ≡  do {
  atomic (do {
  exists ← call (KeyExists u);
    (if exists = Bool True then do {
      call (NestedOp u (Mail (Assign mail)))
    } else skip)
  });
  return Undef
}

definition removeUser-impl :: val ⇒ (val,operation,val) io where
  removeUser-impl u ≡  do {
  atomic (do {
      call (DeleteKey u)
  });
  return Undef
}

definition getUser-impl :: val ⇒ (val,operation,val) io where
  getUser-impl u ≡  do {
  atomic (do {
    exists ← call (KeyExists u);
    (if exists = Bool True then do {
      name ← call (NestedOp u (Name Read));
      mail ← call (NestedOp u (Mail Read));
      return (Found (stringval name) (stringval mail))
    } else return NotFound)
  })
}
```

Figure 8.7.: *Implementation of the user database example in Isabelle.*

```
definition inv1 where
  inv1 op res ihb ≡ ∀ r g u g-res.
    op r ≙ RemoveUser u
  ⟶ op g ≙ GetUser u
  ⟶ (r,g) ∈ ihb
  ⟶ res g ≙ g-res
  ⟶ g-res = NotFound

definition inv2 where
  inv2 c-calls hb ≡ ¬(∃ write delete u upd.
    (cOp c-calls write ≙ NestedOp u upd)
  ∧ is-update upd
  ∧ (cOp c-calls delete ≙ DeleteKey u)
  ∧ (delete, write) ∈ hb
  )

definition inv3 where
  inv3 op i-origin c-calls ≡ ∀ u i c.
    op i ≙ RemoveUser u
  ⟶ i-origin c ≙ i
  ⟶ cOp c-calls c ≙ DeleteKey (UserId u)

definition inv :: (proc, operation, val) invContext ⇒ bool where
  inv ctxt ≡
    inv1 (invocOp ctxt) (invocRes ctxt) (invocation-happensBefore ctxt)
  ∧ inv2 (calls ctxt) (happensBefore ctxt)
  ∧ inv3 (invocOp ctxt) (i-callOriginI ctxt) (calls ctxt)
```

Figure 8.8.: *Invariants for the user database example in Isabelle.*

Procedure RemoveUser

For this procedure, the start of an invocation is more interesting, since *inv1* mentions this procedure. To show that the invariant is preserved, we can use the fact that a new procedure invocation cannot occur before any other invocation existing in the history. The same is true, for the proof obligation when committing the transaction.

We can also easily show that the shape invariant *inv3* holds since the delete is the only call made in the transaction.

Procedure GetUser

The *GetUser* procedure is essential for *inv1*. Directly, after the start of the invocation and after committing the transaction, *inv1* is not yet affected, since there is no result for the invocation yet.

Only after returning *Found* from the invocation, there is an interesting proof obligation for *inv1*. Here, we have to show that there can be no *RemoveUser* invocation before the current invocation that deleted the user. If we return *Found*, the *KeyExists* query must have returned *True*. From this we can derive that there must be an update operation on the user such that all deletes happened before the update. Together with *inv2* we get that there can be no delete operation. However, if there were an invocation of *RemoveUser* before the current invocation, according to *inv3* there should have been a delete operation that is visible. Thus, there can be no such invocation and *inv1* still holds.

8.3. Further Examples

We have modelled a few other examples in Repliss, which we introduce now. These examples were not verified in Isabelle and only analyzed with the Repliss testing or verification tools.

8.3.1. Chat Example Data Invariant

For the chat application described earlier, we have verified the following data invariants in Repliss:

```
invariant forall m: MessageId ::
    chatQry(Contains(m)) ==> messageQry(ContainsKey(m))

invariant forall m: MessageId ::
    messageQry(ContainsKey(m)) ==> chatQry(Contains(m))
```

Together these invariants are a referential integrity property. They state that a message is in the set of all messages (the `chat`) if and only if it has an entry in the `message` map.

We would have liked to express this property as one invariant using the appropiate operator (`<==>`). However, the invariants can then no longer be verified automatically by the SMT solvers, even though both formulations are

```
def store(e: Element)
    atomic
        if sQry(GetSize) < 1
            call s(Add(e))

crdt s: Set_aw[Element]

invariant sQry(GetSize) < 3
```

Figure 8.9.: *Limited set size example.*

logically equivalent. This is one of the examples showing the fragility of SMT solvers in some situations.

8.3.2. Limited Set Size

The application in Figure 8.9 consists of a single procedure. The procedure adds a new element to a set only if the size of the set is less than one. In a strongly consistent setting, this would guarantee that there can never be more than one element in the set. However, in Repliss there is no bound on the size of the set. We use this artificial example, to check whether the automated testing techniques can find bigger examples with several concurrent procedure invocations. The example can be varied by changing the value `3` in the invariant. A bigger value means that the size of the minimal counter example increases. Below, we identify these examples with `singleton_set`N`.rpls`.

8.4. Evaluating Performance

We used the above case studies to evaluate the performance of Repliss. The experiments were evaluated on a Laptop with i7-7500U processor and 16GB of RAM. The software was run on Ubuntu 20.04 with Java OpenJDK version 11.0.8, Z3 version 4.8.9.0, and CVC4 version 1.7. The Benchmark code is available in `Benchmark.scala` in the Repliss sources[1].

We evaluated the performance of three different aspects of the Repliss tool:

1. A comparison of the three different testing techniques we implemented.
2. The time required for symbolic execution.
3. A comparison of the different SMT solvers for individual proof obligations.

8.4.1. Automatic Testing

We evaluated the time to find a counter example with the different testing strategies. These are the following:

[1] `https://github.com/peterzeller/repliss` at commit `93c24670f8`

Example	Random	Small	Dedup
chatapp_fail1.rpls	5.8	4.5	1.1
chatapp_fail2.rpls	3.3	8.2	1.0
userbase_fail1.rpls	3.6	17.5	1.5
userbase_fail2.rpls	1.3	2.3	0.5
singleton_set2.rpls	0.6	0.3	0.2
singleton_set3.rpls	1.0	-	0.2
singleton_set4.rpls	4.3	-	2.7
singleton_set5.rpls	8.5	-	132.2

Figure 8.10.: *Time in seconds for different testing strategies until a bug is found.*

Random Random testing similar to QuickCheck (See Section 7.3.2.)

Small Systematic exploration of small executions similar to SmallCheck (see Section 7.3.4)

Dedup Systematic exploration of small executions with deduplication by checking for equivalent states (see Section 7.3.4)

The results are shown in Figure 8.10 with the different strategies in the columns.

8.4.2. Symbolic Execution

In Figure 8.11 we show the verification time for different examples we verified with Repliss. The running times are subdivided into the running time for the invariant check of the initial state and the running time for checking each procedure. As SMT solver, we used the Repliss default option, which is an incremental solver that concurrently runs two CVC4 instances (one with and one without the finite model finding option). Z3 is not included in the default options, since a bug in Z3 prevents us from interrupting the solver if another solver is faster. However, we evaluate the performance of Z3 separately in the next subsection.

We evaluated the running time for the following examples:

chatapp The chat application as introduced in Chapter 2, but using a manually written shape invariant tailored to the problem.

chatapp_si Same as above, but using the automatically generated shape invariants.

chatapp_data Verifying the data invariant for the chat application(see Section 8.3.1).

userbase The user database example as described in Section 8.2 with manual shape invariants.

userbase2 Same as above, but using the automatically generated shape invariants while keeping one manual shape invariant.

`chatapp.rpls`	`chatapp_si.rpls`	`chatapp_data.rpls`
0.3s initial	0.1s initial	0.2s initial
4.2s sendMessage	4.0s sendMessage	1m 50s sendMessage
3.8s editMessage	9.4s editMessage	2m 53s editMessage
2.7s deleteMessage	10.2s deleteMessage	2m 25s deleteMessage
37.3s getMessage	1m 12s getMessage	39s getMessage
48.5s Σ	1m 36s Σ	7m 48s Σ
`userbase.rpls`	`userbase2.rpls`	`userbase3.rpls`
0.1s initial	0.1s initial	0.1s initial
4.0s registerUser	4.4s registerUser	3.8s registerUser
3.8s updateMail	11.6s updateMail	11.5s updateMail
1.5s removeUser	2.0s removeUser	1.5s removeUser
10.2s getUser	2m 5s getUser	1m 55s getUser
19.7s Σ	2m 34s Σ	2m 12s Σ

Figure 8.11.: *Time for verifying different variants of the case studies.*

userbase3 Same as above, but using the automatically generated shape invariants.

The evaluation shows that the procedures that were harder to verify in Isabelle, are also harder for the SMT solvers. For the chat application this is the `getMessage` procedure and for the user database it is `getUser`.

For the data-invariant of the chat application this situation is reversed and the procedures containing updates use the most times, whereas `getMessage` only performs database queries and is thus easier to verify. The fact that this procedure still takes so long to verify hints at some possible optimizations for handling data invariants. In principle, it should be trivial to verify the data-invariant for a transaction without updates. However, it seems that the SMT solver cannot do this rewriting and then detect the equivalence to the invariant in the pre-state.

It is also apparent from the running times that the automatically generated shape invariants add a significant overhead compared to invariants manually tailored to the problem at hand.

Finding Counter Examples. Besides measuring the time for successful verification attempts, we have also measures the time for symbolic execution to find a counter example. For this, we run the verifier on examples containing genuine bugs and examples where the application is correct but verification fails due to missing invariants. The results are given in Figure 8.12. The two columns *no SI* and *SI* denote whether automatically generated shape invariants were enabled or not.

We can see that with automatic shape invariants, the problem of finding counter examples is much slower. However, without these invariants, counter examples are less intuitive since they may show procedure invocations with database calls that do not appear in the implementation of the procedure.

Example	no SI	SI
chatapp_fail1.rpls	8s	23s
chatapp_fail2.rpls	10s	16m 35s
userbase_fail1.rpls	6s	5m 29s
userbase_fail2.rpls	6s	2m 24s
chatapp1.rpls	9s	23s
chatapp2.rpls	7s	29s
chatapp3.rpls	43s	16m 54s
chatapp_si1.rpls	9s	32s
chatapp_si2.rpls	10s	17m 14s
userbase_si.rpls	6s	8m 17s

Figure 8.12.: *Time until symbolic execution finds counter examples.*

The relatively long times to find a verification counter example also shows the benefit of having an automated testing tool that often can find counter examples much faster if there is a genuine bug.

8.4.3. SMT Solver Performance

During symbolic execution many queries to SMT solvers are generated. We have collected the queries and compare the running times of the Z3 and CVC4 theorem solvers. In the evaluation we distinguish between satisfiable and unsatisfiable problems.

The results for unsatisfiable problems are given in Figure 8.13. The y-axis shows the number of instances for intervals of length 10. The x-axis shows the difference in time to solve the query. For the value 0 both solvers take the time. A negative value indicates that Z3 was faster, a positive value that CVC4 was faster. The value denotes the reduction in time, so a value of 80 means that CVC4 used only 20% of the time required by Z3. Values above 100 mean that the instance could not be solved by the other solver.

We can see that there is an advantage on the side of Z3. In particular, Z3 can proof the data invariant much faster. However, CVC4 could solve more instances in total with the timeout of 1 minute we used for this comparison.

For satisfiable problems, we have a similar comparison in Figure 8.14. Here, we used the finite model finding option of CVC4. This option is made for solving satisfiable instances and the evaluation results show that this gives some significant advantages. CVC4 is faster and can solve many problems for which Z3 does not find a counter example in the given timeout.

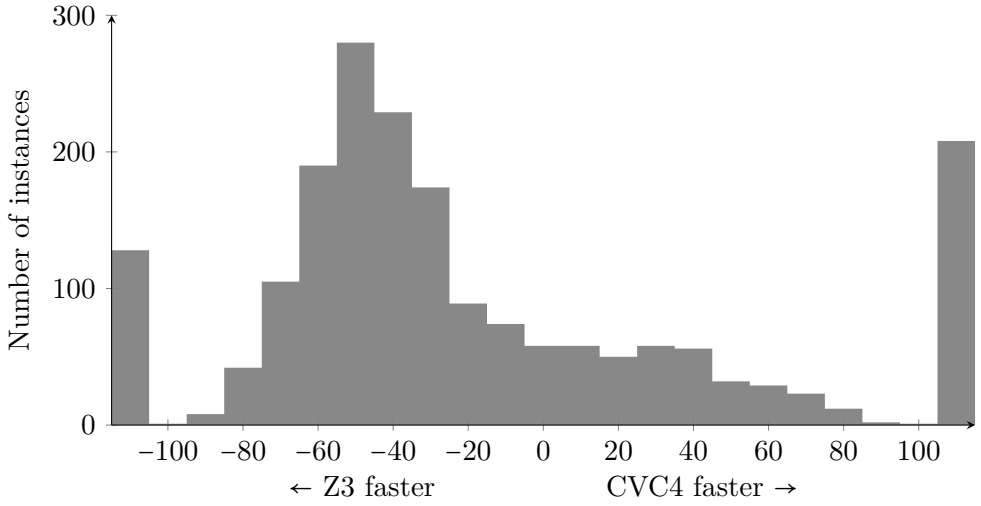


Figure 8.13.: *Comparison of Z3 and CVC4 for UNSAT problems.*

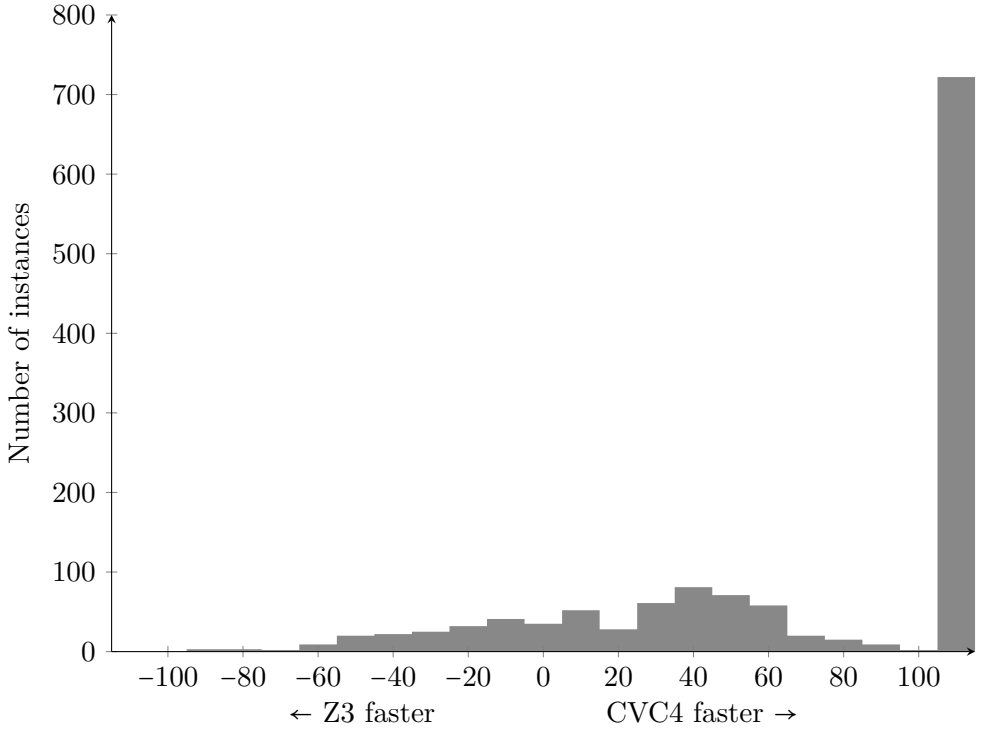


Figure 8.14.: *Comparison of Z3 and CVC4 with finite model finder option for SAT problems.*

Chapter 9

Conclusions

In this thesis, we have presented a new technique for verifying highly available applications. The technique is enabled by three main design choices:

1. Communication between concurrent procedure invocation is limited to updating the shared replicated database.
2. Invariants are restricted to address the global state.
3. Specifications are based on event histories.

The first two points are essential to allow for local reasoning. By restricting communication to the database, which has a well-defined semantics, we also limit where possible interactions between concurrent processes can occur. This makes it possible to reason locally except for a few program points. For these points, we use global invariants to reason about the possible effects of concurrent processes.

Our specification technique based on event histories poses new challenges when it comes to the development of tools which support partially automated verification. We have developed the Repliss tool to explore how well our specification and verification technique is suitable for automation. The specification technique based on event histories distinguishes Repliss from other verification tools. The Repliss tool demonstrates some advantages of this specification technique. The specifications are expressive and allow to specify some properties that could only be specified using ghost variables in state based techniques. It also improves our approach to handle concurrency. Since the history always grows monotonically, we have an implicit restriction on what can happen in concurrently executed code. Another advantage of using the history instead of states, is that counter examples arising from automated provers can be more intuitive. Instead of a single state, a counter example is given as a partial execution. On the other hand, relying on history information instead of abstract state, complicates some verification conditions.

Overall, we showed that our technique is suitable for handling the special kind of concurrency prevalent in highly available applications. Our work also

is a sizable example of completely verifying the soundness of a proof technique in Isabelle/HOL. This effort is based on a small-step interleaving semantics. It then continues with a reduction to the single-invocation semantics that eliminates the main pain points of concurrency. Moreover, the Isabelle theories include specific proof rules that enable verification applications using a kind of symbolic execution, resulting in a framework that can be used within Isabelle. Our case studies for two examples demonstrate that this verification in Isabelle works using equivalent invariants to the ones used for verification with the Repliss tool.

9.1. Future Research

Closing Formal Gaps. We have formalized a significant part of our technique in Isabelle/HOL, starting from a semantics and leading up to proof rules for symbolic execution. For this part, we can be certain of correctness of all theorems, assuming no Isabelle/HOL bugs affecting this result.

However, around these core results, there are some parts we have not proven formally. As highlighted by Fonseca et al. [Fon+17], these untrusted parts often contain bugs. Thus, we list the untrusted parts of our development in the following and discuss how the gaps could be closed.

1. Our semantics constitute the basis for all formalizations. However, we have not verified that this semantics accurately describes actual database systems. One could close this gap formally, by implementing a database system in Isabelle/HOL and then verifying that it satisfies our semantics when combined with an application program. Similar efforts already have been made. For example IronFleet [Haw+17] is an effort to fully verify a distributed database, but it has different semantics from our setting. In particular, the database does not provide replicated data types. In this thesis, we only worked with specifications of replicated data types. However, our earlier work on verifying CRDTs in Isabelle/HOL [ZBP14a] could be combined with this thesis to close this gap.

2. While we have developed and verified proof rules for symbolic execution in Isabelle/HOL, we did not verify that the realization of these rules in the Repliss tool is correct. If we wanted to close this gap, we could implement the Repliss tool in Isabelle/HOL, where we could then prove the correspondence to the proof rules. Then Isabelle's code generation could be used to get the executable Repliss tool. One challenge in doing so, is that Repliss invokes SMT solvers as external tools, so we would also require a formalization of their semantics. Doing this correctly can again be challenging. For example, while implementing Repliss, we stumbled upon the fact that sets in CVC4 use a non-standard model, where the universe set not necessarily contains all elements.

3. We have not implemented a method to get an executable application

from a Repliss model. Thus, applying Repliss in the development of a highly available application currently would require a manual translation step, which could introduce bugs. This gap could be closed by implementing code generation. Related work like CompCert [Ler+16] or work by Blech [Ble09] shows that it is feasible to build a verified compiler.

Support for different consistency models. In our semantics, we have fixed the consistency model to be transactional causal consistency. We fixed the consistency model to keep the semantics easy to manage and understand, which would change with a model supporting multiple consistency levels. However, there is no fundamental reason why our approach could not be extended to other consistency levels, as long as these consistency levels can be described as constraints on database histories (database calls and happens-before relation).

For supporting a weaker consistency level, one would need to remove some premises from the proof rules, which (as expected) would make some applications more difficult (or impossible) to prove correct. Also, some equivalences for CRDT specifications would be no longer valid without causal consistency.

Support for stronger consistency levels is already possible in a limited way. For example to support locks, we could define a CRDT with a `lock` operation. Then we could specify, as a precondition for invariants, that two transactions invoking the `lock` operation can never be concurrent. However, this extension would not provide any higher level reasoning support for locks. For supporting this, it would be interesting to study an integration of the CISE [Got+16] technique into ours (see related work in Section 3.3). This technique follows the idea of rely-guarantee style reasoning. It can be seen as a way to make our *state-monotonicGrowth* predicate more precise and specific to the locks acquired by a transaction. Basically, we could add two-state invariants associated with locks that each relates an old and a new state and may only be violated by transactions that acquire the related locks. Thus, at the start of a transaction, we can then assume the invariant for the locks excluded by the locks of the started transaction. At the end of the transaction, we then have to show all invariants except for the ones for which we acquired the locks.

Combining interactive and automatic verification. In its current state, the Repliss tool only has limited support for interactive verification. When the automatic SMT solvers fail to verify a proof obligation, it is possible to export the proof obligation to Isabelle, where it can be proved interactively. However, there is no way for the automatic theorem prover to benefit from the interactive proof. For example, this could be achieved by learning auxiliary lemmata and typical instantiations of quantifiers. In particular, this could be helpful for complex functions like aggregates. These are not directly supported by SMT solvers, so they have to be implemented using uninterpreted functions and appropriate axioms and triggers. A good example for this process is the handling of array comprehensions in Dafny [LM09]. We have not yet implemented aggregates for Repliss, so datatypes like counters that require aggregates cannot be verified automatically. In contrast, a higher order

tool like Isabelle/HOL allows to fully define this kind of functions and auxiliary theorems can be proven by methods like induction. Thus, it would be interesting to see, whether an automated verification tool could be based on a higher order logic with the possibility to fall back to interactive verification when new situations with insufficient auxiliary theorems arise.

Bibliography

[AEM17] Hagit Attiya, Faith Ellen, and Adam Morrison. "Limitations of Highly-Available Eventually-Consistent Data Stores". In: *IEEE Trans. Parallel Distrib. Syst.* 28.1 (2017), pp. 141–155. DOI: 10.1109/TPDS.2016.2556669. URL: https://doi.org/10.1109/TPDS.2016.2556669 (cited on page 16).

[AS85] Bowen Alpern and Fred B. Schneider. "Defining Liveness". In: *Inf. Process. Lett.* 21.4 (1985), pp. 181–185. DOI: 10.1016/0020-0190(85)90056-0. URL: https://doi.org/10.1016/0020-0190(85)90056-0 (cited on page 57).

[ASB18] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Delta state replicated data types". In: *J. Parallel Distributed Comput.* 111 (2018), pp. 162–173. DOI: 10.1016/j.jpdc.2017.08.003. URL: https://doi.org/10.1016/j.jpdc.2017.08.003 (cited on page 25).

[Ash75] Edward A. Ashcroft. "Proving Assertions about Parallel Programs". In: *J. Comput. Syst. Sci.* 10.1 (1975). DOI: 10.1016/S0022-0000(75)80018-3. URL: http://dx.doi.org/10.1016/S0022-0000(75)80018-3 (cited on page 18).

[ASS13] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. "Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems". In: *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*. IEEE Computer Society, 2013, pp. 163–172. ISBN: 978-0-7695-5115-9. DOI: 10.1109/SRDS.2013.25. URL: https://doi.org/10.1109/SRDS.2013.25 (cited on page 17).

[Att+16] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. "Specification and Complexity of Collaborative Text Editing". In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*. Ed. by George Giakkoupis. ACM, 2016, pp. 259–268. DOI: 10.1145/2933057.2933090. URL: https://doi.org/10.1145/2933057.2933090 (cited on page 26).

[Baq+17] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. "Composition in State-based Replicated Data Types". In: *Bulletin of the EATCS* 123 (2017). URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/507 (cited on pages 33, 34).

[Bar+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. 2005. DOI: 10.1007/11804192_17. URL: http://dx.doi.org/10.1007/11804192_17 (cited on page 142).

[Bar+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. ISBN: 978-3-642-22109-5. DOI: 10.1007/978-3-642-22110-1_14. URL: https://doi.org/10.1007/978-3-642-22110-1%5C_14 (cited on page 17).

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016 (cited on page 131).

[Bla+14] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. "Truly Modular (Co)datatypes for Isabelle/HOL". In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Ed. by Gerwin Klein and Ruben Gamboa. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014, pp. 93–110. ISBN: 978-3-319-08969-0. DOI: 10.1007/978-3-319-08970-6_7. URL: https://doi.org/10.1007/978-3-319-08970-6%5C_7 (cited on page 92).

[Ble09] Jan Olaf Blech. "Certifying system translations using higher order theorem provers". PhD thesis. University of Kaiserslautern, 2009. ISBN: 978-3-8325-2211-7. URL: http://d-nb.info/994725647 (cited on page 171).

[BM03] Bernhard Beckert and Wojciech Mostowski. "A Program Logic for Handling JAVA CARD's Transaction Mechanism". In: *Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by Mauro Pezzè.

Vol. 2621. Lecture Notes in Computer Science. Springer, 2003, pp. 246–260. ISBN: 3-540-00899-3. DOI: 10.1007/3-540-36578-8_18. URL: https://doi.org/10.1007/3-540-36578-8_18 (cited on page 20).

[BM16] Rastislav Bodík and Rupak Majumdar, eds. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016. ISBN: 978-1-4503-3549-2. URL: http://dl.acm.org/citation.cfm?id=2837614.

[Bou92] Raymond T. Boute. "The Euclidean Definition of the Functions Div and Mod". In: *ACM Trans. Program. Lang. Syst.* 14.2 (Apr. 1992), pp. 127–144. ISSN: 0164-0925. DOI: 10.1145/128861.128862. URL: https://doi.org/10.1145/128861.128862 (cited on page 131).

[Bro+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, et al. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. Ed. by Andrew Birrell and Emin Gün Sirer. USENIX Association, 2013, pp. 49–60. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson (cited on page 15).

[Bul+08] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. "Imperative Functional Programming with Isabelle/HOL". In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Ed. by Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170. Lecture Notes in Computer Science. Springer, 2008, pp. 134–149. ISBN: 978-3-540-71065-3. DOI: 10.1007/978-3-540-71067-7_14. URL: https://doi.org/10.1007/978-3-540-71067-7%5C_14 (cited on page 86).

[Bul12] Lukas Bulwahn. "The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof". In: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*. Ed. by Chris Hawblitzel and Dale Miller. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 92–108. DOI: 10.1007/978-3-642-35308-6_10. URL: https://doi.org/10.1007/978-3-642-35308-6%5C_10 (cited on page 134).

[Bur+14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. "Replicated data types: specification, verification, optimality". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagan-

nathan and Peter Sewell. ACM, 2014, pp. 271–284. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535848. URL: http://doi.acm.org/10.1145/2535838.2535848 (cited on pages 19, 26).

[Bur14] Sebastian Burckhardt. "Principles of Eventual Consistency". In: *Foundations and Trends in Programming Languages* 1.1-2 (2014), pp. 1–150. DOI: 10.1561/2500000011. URL: https://doi.org/10.1561/2500000011 (cited on page 26).

[CBG15] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. "A Framework for Transactional Consistency Models with Atomic Visibility". In: *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. Ed. by Luca Aceto and David de Frutos-Escrig. Vol. 42. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 58–71. ISBN: 978-3-939897-91-0. DOI: 10.4230/LIPIcs.CONCUR.2015.58. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2015.58 (cited on page 17).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973 (cited on page 122).

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011. ISBN: 978-3-642-15259-7. DOI: 10.1007/978-3-642-15260-3. URL: https://doi.org/10.1007/978-3-642-15260-3 (cited on page 56).

[Chu36] Alonzo Church. "A Note on the Entscheidungsproblem". In: *The Journal of Symbolic Logic* 1.1 (1936), pp. 40–41. ISSN: 00224812. URL: http://www.jstor.org/stable/2269326 (cited on page 133).

[Coo15] James Cook. https://hackage.haskell.org/package/monad-loops. Accessed: 2019-02-20. June 2015 (cited on page 89).

[Coo78] Stephen A. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In: *SIAM J. Comput.* 7.1 (1978), pp. 70–90. DOI: 10.1137/0207005. URL: https://doi.org/10.1137/0207005 (cited on page 123).

[Cor+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, et al. "Spanner: Google's Globally-Distributed Database". In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012.*

Ed. by Chandu Thekkath and Amin Vahdat. USENIX Association, 2012, pp. 251–264. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett (cited on page 16).

[DD15] Brijesh Dongol and John Derrick. "Verifying Linearisability: A Comparative Survey". In: *ACM Comput. Surv.* 48.2 (2015), 19:1–19:43. DOI: 10.1145/2796550. URL: https://doi.org/10.1145/2796550 (cited on page 19).

[DO14] Crystal Chang Din and Olaf Owe. "A sound and complete reasoning system for asynchronous communication with shared futures". In: *J. Log. Algebr. Meth. Program.* 83.5-6 (2014), pp. 360–383. DOI: 10.1016/j.jlamp.2014.03.003. URL: https://doi.org/10.1016/j.jlamp.2014.03.003 (cited on page 19).

[Doh+18] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. "Making Linearizability Compositional for Partially Ordered Executions". In: *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*. Ed. by Carlo A. Furia and Kirsten Winter. Vol. 11023. Lecture Notes in Computer Science. Springer, 2018, pp. 110–129. DOI: 10.1007/978-3-319-98938-9_7. URL: https://doi.org/10.1007/978-3-319-98938-9%5C_7 (cited on page 18).

[EG89] Clarence A. Ellis and Simon J. Gibbs. "Concurrency Control in Groupware Systems". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*. Ed. by James Clifford, Bruce G. Lindsay, and David Maier. ACM Press, 1989, pp. 399–407. DOI: 10.1145/67544.66963. URL: https://doi.org/10.1145/67544.66963 (cited on page 16).

[Ene+19] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. "Efficient Synchronization of State-Based CRDTs". In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 148–159. DOI: 10.1109/ICDE.2019.00022. URL: https://doi.org/10.1109/ICDE.2019.00022 (cited on page 25).

[Fil11] Jean-Christophe Filliâtre. "Deductive software verification". In: *International Journal on Software Tools for Technology Transfer* 13.5 (Aug. 2011), p. 397. ISSN: 1433-2787. DOI: 10.1007/s10009-011-0211-0. URL: https://doi.org/10.1007/s10009-011-0211-0 (cited on page 2).

[FL17] Richard L Ford and K Rustan M Leino. *Dafny Reference Manual*. Available at https://github.com/dafny-lang/dafny/blob/131b88f7a0d43f5c4eb811395f0d7baa3065780e/Docs/DafnyRef/out/DafnyRef.pdf. 2017 (cited on pages 117, 124).

[Flo93] Robert W Floyd. "Assigning Meanings to Programs". In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R Colburn, James H Fetzer, and Terry L Rankin. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4. URL: https://doi.org/10.1007/978-94-011-1793-7_4 (cited on page 117).

[Fon+17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. "An Empirical Study on the Correctness of Formally Verified Distributed Systems". In: *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic. ACM, 2017, pp. 328–343. DOI: 10.1145/3064176.3064183. URL: https://doi.org/10.1145/3064176.3064183 (cited on page 170).

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". In: *Proceedings of the 22nd European Symposium on Programming*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, Mar. 2013, pp. 125–128 (cited on page 142).

[Fut+85] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. "Principles of OBJ2". In: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*. Ed. by Mary S. Van Deusen, Zvi Galil, and Brian K. Reid. ACM Press, 1985, pp. 52–66. DOI: 10.1145/318593.318610. URL: https://doi.org/10.1145/318593.318610 (cited on page 20).

[GC10] Mike Gordon and Hélène Collavizza. "Forward with Hoare". In: *Reflections on the Work of C. A. R. Hoare*. Ed. by A. W. Roscoe, Clifford B. Jones, and Kenneth R. Wood. Springer, 2010, pp. 101–121. ISBN: 978-1-84882-911-4. DOI: 10.1007/978-1-84882-912-1_5. URL: https://doi.org/10.1007/978-1-84882-912-1%5C_5 (cited on page 97).

[GL02] Seth Gilbert and Nancy A. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (2002) (cited on page 1).

[Got+16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. "'Cause I'm strong enough: reasoning about consistency choices in distributed systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 371–384. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837625. URL: http://doi.acm.org/10.1145/2837614.2837625 (cited on pages 19, 171).

[Gri+09] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. "Strongest postcondition of unstructured programs". In: *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs, FTfJP 2009, Genova, Italy, July 6, 2009*. Ed. by Anindya Banerjee. ACM, 2009, 6:1–6:7. ISBN: 978-1-60558-540-6. DOI: 10.1145/1557898.1557904. URL: https://doi.org/10.1145/1557898.1557904 (cited on page 97).

[Gri82] David Gries. "A Note on a Standard Strategy for Developing Loop Invariants and Loops". In: *Sci. Comput. Program.* 2.3 (1982), pp. 207–214. DOI: 10.1016/0167-6423(83)90015-1. URL: https://doi.org/10.1016/0167-6423(83)90015-1 (cited on page 115).

[GY15a] Alexey Gotsman and Hongseok Yang. "Composite Replicated Data Types". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP*. 2015. DOI: 10.1007/978-3-662-46669-8_24. URL: http://dx.doi.org/10.1007/978-3-662-46669-8_24 (cited on pages 19, 26).

[GY15b] Alexey Gotsman and Hongseok Yang. "Composite Replicated Data Types". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 585–609. ISBN: 978-3-662-46668-1. DOI: 10.1007/978-3-662-46669-8_24. URL: https://doi.org/10.1007/978-3-662-46669-8_24 (cited on pages 19, 26, 33, 34).

[Haw+17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: proving safety and liveness of practical distributed systems". In: *Commun. ACM* 60.7 (2017), pp. 83–92. DOI: 10.1145/3068608. URL: https://doi.org/10.1145/3068608 (cited on page 170).

[Hin69] Roger Hindley. "The principal type-scheme of an object in combinatory logic". In: *Transactions of the american mathematical society* 146 (1969), pp. 29–60 (cited on page 21).

[Jac+05] Bart Jacobs, Erik Meijer, Frank Piessens, and Wolfram Schulte. "Iterators Revisited: Proof Rules and Implementation". In: *IN WORKSHOP ON FORMAL TECHNIQUES FOR JAVA-LIKE PROGRAMS (FTFJP*. 2005 (cited on page 115).

[Jac+08] Bart Jacobs, Frank Piessens, Jan Smans, K. Rustan M. Leino, and Wolfram Schulte. "A programming model for concurrent object-oriented programs". In: *ACM Trans. Program. Lang. Syst.* 31.1

(2008), 1:1–1:48. DOI: 10.1145/1452044.1452045. URL: https://doi.org/10.1145/1452044.1452045 (cited on page 18).

[Jac+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55. ISBN: 978-3-642-20397-8. DOI: 10.1007/978-3-642-20398-5_4. URL: https://doi.org/10.1007/978-3-642-20398-5%5C_4 (cited on page 18).

[Jon83] Cliff B. Jones. "Specification and Design of (Parallel) Programs". In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Ed. by R. E. A. Mason. North-Holland/IFIP, 1983, pp. 321–332. ISBN: 0-444-86729-5 (cited on page 18).

[Kak+18] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. "Safe Replication through Bounded Concurrency Verification". In: *32nd ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 2018 (cited on page 20).

[KMS12] Ioannis T. Kassios, Peter Müller, and Malte Schwerhoff. "Comparing Verification Condition Generation with Symbolic Execution: An Experience Report". In: *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*. Ed. by Rajeev Joshi, Peter Müller, and Andreas Podelski. Vol. 7152. Lecture Notes in Computer Science. Springer, 2012, pp. 196–208. ISBN: 978-3-642-27704-7. DOI: 10.1007/978-3-642-27705-4_16. URL: https://doi.org/10.1007/978-3-642-27705-4%5C_16 (cited on pages 97, 140).

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X (cited on page 19).

[Lam98] Leslie Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. DOI: 10.1145/279227.279229. URL: https://doi.org/10.1145/279227.279229 (cited on page 16).

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. "Chapar: certified causally consistent distributed key-value stores". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav

Bodík and Rupak Majumdar. ACM, 2016, pp. 357–370. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837622. URL: http://doi.acm.org/10.1145/2837614.2837622 (cited on page 20).

[Lei20] K. Rustan M. Leino. *Dafny Power User: old and unchanged*. http://leino.science/papers/krml273.html (published on 15 February 2020). 2020 (cited on page 124).

[Ler+16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert - A Formally Verified Optimizing Compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE. Toulouse, France, Jan. 2016. URL: https://hal.inria.fr/hal-01238879 (cited on page 171).

[Lew+19] Nicholas V. Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Cerný. "Sequential programming for replicated data stores". In: *PACMPL* 3.ICFP (2019), 106:1–106:28. DOI: 10.1145/3341710. URL: https://doi.org/10.1145/3341710 (cited on page 20).

[Lip75] Richard J. Lipton. "Reduction: A Method of Proving Properties of Parallel Programs". In: *Commun. ACM* 18.12 (1975), pp. 717–721. DOI: 10.1145/361227.361234. URL: https://doi.org/10.1145/361227.361234 (cited on page 63).

[Llo+11] 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 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. Ed. by Ted Wobber and Peter Druschel. ACM, 2011, pp. 401–416. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043593. URL: http://doi.acm.org/10.1145/2043556.2043593 (cited on pages 1, 16).

[LM09] K. Rustan M. Leino and Rosemary Monahan. "Reasoning about comprehensions with first-order SMT solvers". In: *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*. Ed. by Sung Y. Shin and Sascha Ossowski. ACM, 2009, pp. 615–622. ISBN: 978-1-60558-166-8. DOI: 10.1145/1529282.1529411. URL: https://doi.org/10.1145/1529282.1529411 (cited on page 171).

[LM10] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922 (cited on page 7).

[LMS09] Rustan Leino, Peter Müller, and Jan Smans. "Verification of Concurrent Programs with Chalice". In: *Foundations of Security Analysis and Design V*. June 2009, pp. 195–222. URL: https://www.microsoft.com/en-us/research/publication/verification-concurrent-programs-chalice/ (cited on page 18).

[Mat70] Yuri Vladimirovich Matiyasevich. "The Diophantineness of enumerable sets". In: *Doklady Akademii Nauk*. Vol. 191. 2. Russian Academy of Sciences. 1970, pp. 279–282 (cited on page 133).

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*. 2008. DOI: 10.1007/978-3-540-78800-3_24. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_24 (cited on page 17).

[Mil78] Robin Milner. "A Theory of Type Polymorphism in Programming". In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4. URL: https://doi.org/10.1016/0022-0000(78)90014-4 (cited on page 21).

[Mit20] Neil Mitchell. https://hackage.haskell.org/package/extra. Accessed: 2019-02-20. Feb. 2020 (cited on page 89).

[MMW16] Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. "Eisbach: A Proof Method Language for Isabelle". In: *J. Autom. Reasoning* 56.3 (2016), pp. 261–282. DOI: 10.1007/s10817-015-9360-2. URL: https://doi.org/10.1007/s10817-015-9360-2 (cited on pages 86, 117).

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by B. Jobstmann and K. R. M. Leino. Vol. 9583. LNCS. Springer-Verlag, 2016, pp. 41–62 (cited on page 18).

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990. ISBN: 978-0-262-63132-7 (cited on page 20).

[Naj+16] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. "The CISE tool: proving weakly-consistent applications correct". In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*. Ed. by Peter Alvaro and Alysson Bessani. ACM, 2016, 2:1–2:3. ISBN: 978-1-4503-4296-4. DOI: 10.1145/2911151.2911160. URL: http://doi.acm.org/10.1145/2911151.2911160 (cited on page 19).

[Nip06] Tobias Nipkow. "Abstract Hoare Logics". In: *Archive of Formal Proofs* (Aug. 2006). http://isa-afp.org/entries/Abstract-Hoare-Logics.html, Formal proof development. ISSN: 2150-914x (cited on pages 123, 124).

[NPS19] Sreeja Nair, Gustavo Petri, and Marc Shapiro. "Invariant Safety for Distributed Applications". In: *CoRR* abs/1903.02759 (2019). arXiv: 1903.02759. URL: http://arxiv.org/abs/1903.02759 (cited on page 19).

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002 (cited on pages 20, 49, 88).

[OG76] Susan S. Owicki and David Gries. "An Axiomatic Proof Technique for Parallel Programs I". In: *Acta Informatica* 6 (1976), pp. 319–340. DOI: 10.1007/BF00268134. URL: https://doi.org/10.1007/BF00268134 (cited on page 18).

[OO14] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro (cited on page 16).

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013 (cited on page 126).

[Pau19] Lawrence C. Paulson. "Zermelo Fraenkel Set Theory in Higher-Order Logic". In: *Archive of Formal Proofs* 2019 (2019). URL: https://www.isa-afp.org/entries/ZFC%5C_in%5C_HOL.html (cited on page 91).

[Pel01] Doron A. Peled. *Software Reliability Methods*. Texts in Computer Science. Springer, 2001. ISBN: 978-1-4419-2876-4. DOI: 10.1007/978-1-4757-3540-6. URL: http://u.cs.biu.ac.il/%5C%7Edoronp/srm.html (cited on page 17).

[Pey03] Simon L. Peyton Jones. "Haskell 98: Expressions". In: *J. Funct. Program.* 13.1 (2003), pp. 17–38. DOI: 10.1017/S0956796803000510. URL: https://doi.org/10.1017/S0956796803000510 (cited on page 88).

[Pre18] Nuno M. Preguiça. "Conflict-free Replicated Data Types: An Overview". In: *CoRR* abs/1806.10254 (2018). arXiv: 1806.10254. URL: http://arxiv.org/abs/1806.10254 (cited on pages 25, 33, 34).

[Rey+13] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark Barrett. "Quantifier Instantiation Techniques for Finite Model Finding in SMT". In: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*. Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, 2013, pp. 377–391. ISBN: 978-3-642-38573-5. DOI: 10.1007/978-3-642-38574-2_26. URL: https://doi.org/10.1007/978-3-642-38574-2%5C_26 (cited on page 143).

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9. DOI: 10.1109/LICS.2002.1029817. URL: https://doi.org/10.1109/LICS.2002.1029817 (cited on page 18).

[RNL08] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 37–48. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411292. URL: https://doi.org/10.1145/1411286.1411292 (cited on pages 134, 136).

[Sch08] Norbert Schirmer. "A Sequential Imperative Programming Language Syntax, Semantics, Hoare Logics and Verification Environment". In: *Archive of Formal Proofs* (Feb. 2008). http://isa-afp.org/entries/Simpl.html, Formal proof development. ISSN: 2150-914x (cited on page 124).

[Sch16] Malte H Schwerhoff. "Advancing Automated, Permission-Based Program Verification Using Symbolic Execution". PhD thesis. ETH Zurich, 2016 (cited on page 85).

[Ser20a] Amazon Web Services. *Amazon S3 data consistency model*. https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html. Accessed: 2020-09-02. 2020 (cited on page 16).

[Ser20b] Amazon Web Services. *Using versioning - Amazon Simple Storage Service*. https://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html. Accessed: 2020-09-02. 2020 (cited on page 16).

[Sha+11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Anglais. Rapport de recherche RR-7506. INRIA, Jan. 2011. URL: http://hal.inria.fr/inria-00555588 (cited on pages 8, 25, 33, 34).

[Sha+11b] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. Vol. 6976. Lecture Notes in Computer Science. Springer, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29. URL: https://doi.org/10.1007/978-3-642-24550-3%5C_29 (cited on pages 6, 8, 16, 25).

[SKJ15] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. "Declarative programming over eventually consistent data stores". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 413–424. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737981. URL: http://doi.acm.org/10.1145/2737924.2737981 (cited on page 19).

[Sta96] EBNF Syntaxt Specification Standard. "Ebnf: Iso/iec 14977: 1996 (e)". In: 70 (1996). URL: https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf (cited on page 126).

[Ter+94] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*. IEEE Computer Society, 1994, pp. 140–149. ISBN: 0-8186-6400-2. DOI: 10.1109/PDIS.1994.331722. URL: http://dx.doi.org/10.1109/PDIS.1994.331722 (cited on page 16).

[Tur36] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5 (cited on page 133).

[Vaf08] Viktor Vafeiadis. "Modular fine-grained concurrency verification". PhD thesis. University of Cambridge, UK, 2008. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221 (cited on page 18).

[Vis] Stanislav Vishnevskiy. *How Discord Stores Billions of Messages*. https://blog.discordapp.com/how-discord-stores-billions-of-messages-7fa6ec7ee4c7. Accessed: 2018-11-16 (cited on pages 7, 10).

[VV16] Paolo Viotti and Marko Vukolic. "Consistency in Non-Transactional Distributed Storage Systems". In: *ACM Comput. Surv.* 49.1 (2016), 19:1–19:34. DOI: 10.1145/2926965. URL: https://doi.org/10.1145/2926965 (cited on page 16).

[Wad90] Philip Wadler. "Comprehending Monads". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. ACM, 1990, pp. 61–78. ISBN: 0-89791-368-X. DOI: 10.1145/91556.91592. URL: https://doi.org/10.1145/91556.91592 (cited on page 87).

[Wan91] Mitchell Wand. "Type Inference for Record Concatenation and Multiple Inheritance". In: *Inf. Comput.* 93.1 (1991), pp. 1–15. DOI: 10.1016/0890-5401(91)90050-C. URL: https://doi.org/10.1016/0890-5401(91)90050-C (cited on page 21).

[WB89] Philip Wadler and Stephen Blott. "How to Make ad-hoc Polymorphism Less ad-hoc". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 1989, pp. 60–76. DOI: 10.1145/75277.75283. URL: https://doi.org/10.1145/75277.75283 (cited on pages 21, 23).

[Wen+19] Makarius Wenzel, Clemens Ballarin, Stefan Berghofer, Jasmin Blanchette, et al. *The isabelle/isar reference manual*. Available at https://isabelle.in.tum.de/doc/isar-ref.pdf. 2019 (cited on page 117).

[WN04] Martin Wildmoser and Tobias Nipkow. "Certifying Machine Code Safety: Shallow Versus Deep Embedding". In: *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*. Ed. by Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan. Vol. 3223. Lecture Notes in Computer Science. Springer, 2004, pp. 305–320. ISBN: 3-540-23017-3. DOI: 10.1007/978-3-540-30142-4_22. URL: https://doi.org/10.1007/978-3-540-30142-4%5C_22 (cited on page 86).

[WSR00] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. "Shape Analysis". In: *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*. Ed. by David A. Watt. Vol. 1781. Lecture Notes in Computer Science. Springer, 2000, pp. 1–17. ISBN: 3-540-67263-X. DOI: 10.1007/3-540-46423-9_1. URL: https://doi.org/10.1007/3-540-46423-9%5C_1 (cited on page 147).

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications". In: *Correct Hardware Design and Verification Methods*. Ed. by Laurence Pierre and Thomas Kropf. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66. ISBN: 978-3-540-48153-9 (cited on page 137).

[ZBP14a] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. "Formal Specification and Verification of CRDTs". In: *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*. 2014. DOI: 10.1007/978-3-662-43613-4_3. URL: http://dx.doi.org/10.1007/978-3-662-43613-4_3 (cited on page 170).

[ZBP14b] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. "Formal Specification and Verification of CRDTs". In: *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*. Ed. by Erika Ábrahám and Catuscia Palamidessi. Vol. 8461. Lecture Notes in Computer Science. Springer, 2014, pp. 33–48. ISBN: 978-3-662-43612-7. DOI: 10.1007/978-3-662-43613-4_3. URL: https://doi.org/10.1007/978-3-662-43613-4_3 (cited on page 26).

Appendix A

Curriculum Vitae

Education

1998-2007	**Abitur**, *Stefan-George-Gymnasium*, Bingen am Rhein.
2008-2011	**Computer Science, Bachelor**, *TU Kaiserslautern.*
2011-2013	**Computer Science, Master**, *TU Kaiserslautern.*

Bachelor thesis

title	*Modellierung einer Web-Applikation mit der Spezifikationssprache ABS*
supervisors	Dr.-Ing. Yannick Welsch, Dr.-Ing. Jan Schäfer, Prof. Dr. Arnd Poetzsch-Heffter
description	Evaluated the modelling and programming language "ABS", by implementing a case study of a lecture and exercise management system.

Master thesis

title	*Specification and Verification of Convergent Replicated Data Types*
supervisors	Dr. rer. nat. Annette Bieniusa, Prof. Dr. Arnd Poetzsch-Heffter
description	Used Isabelle/HOL to formally verify replicated data types.

Employment History

2007-2008	**Zivildienst**, *Diakonie Krankenhaus*, Bad Kreuznach.
2010,2011	**Tutor for Bachelor course "Logic"**, *TU Kaiserslautern.*

2011-2013 **Wissenschaftliche Hilfskraft**, *Software Technology Group*, TU Kaiserslautern.

- Supervised by Yannick Welsch.
- Maintained and improved compiler and Eclipse-plugin of the ABS language as part of the EU project HATS.
- Worked on the BCVerifier tool for checking backwards-compatibility of Java libraries. In particular worked on the invariant specification language used in the tool.

2014-2020 **Wissenschaftlicher Mitarbeiter**, *Software Technology Group*, TU Kaiserslautern.

- Worked on verification and programming models as part of the EU projects Syncfree and LightKone.
- Teaching assistant for courses:
  - Software-Entwicklung I (2014, 2015, 2016, 2017)
  - Grundlagen der Programmierung (2018, 2019)
  - Programmierpraktikum (2019)
  - Logik (2017, 2018)
  - Specification and Verification with Higher-Order Logic (2014)
  - Compiler and Language-Processing Tools (2015, 2016, 2018)
  - Programming Distributed Systems (2018, 2019)
  - Supervised several seminars, projects and thesis

Appendix B

Publications

The following publications are directly related to this thesis:

- Peter Zeller, Arnd Poetzsch-Heffter: Towards a Proof Framework for Information Systems with Weak Consistency. Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Vienna, Austria.
- Peter Zeller: Testing properties of weakly consistent programs with Repliss. Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2017, Belgrade, Serbia.
- Peter Zeller, Annette Bieniusa, Arnd Poetzsch-Heffter: Combining State- and event-based semantics to verify highly available programs. 16th International Conference on Formal Aspects of Component Software, FACS 2019, Amsterdam, Netherlands.

Further publications not directly related to this thesis:

- Peter Zeller, Annette Bieniusa, Arnd Poetzsch-Heffter: Formal Specification and Verification of CRDTs. Formal Techniques for Distributed Objects, Components, and Systems, FORTE 2014, Berlin, Germany.
- Gonçalo Tomás, Peter Zeller, Valter Balegas, Deepthi Devaki Akkoorath, Annette Bieniusa, João Leitão, Nuno M. Preguiça: FMKe: a Real-World Benchmark for Key-Value Data Stores. Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2017, Belgrade, Serbia.
- Marc Shapiro, Annette Bieniusa, Peter Zeller, Gustavo Petri: Ensuring referential integrity under causal consistency. Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2018, Porto, Portugal.

- Annette Bieniusa, Peter Zeller, Shraddha Barke: Collaborative Work Management with a Highly-Available Kanban Board. Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday, 2018, Kaiserslautern, Germany.
- Peter Zeller, Annette Bieniusa, Carla Ferreira: Teaching practical realistic verification of distributed algorithms in Erlang with TLA+, Erlang Workshop 2020, Virtual Event