Bringing Hybrid Consistency Closer to Programmers

Gonçalo Marcelino
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa
Portugal

Valter Balegas
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa
Portugal

Carla Ferreira
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa
Portugal

ABSTRACT

Hybrid consistency is a new consistency model that tries to combine the benefits of weak and strong consistency. To implement hybrid consistency, programmers have to identify conflicting operations in applications and instrument them, which is a difficult and error prone task. More recent approaches automatize the process through the use of static analysis over a specification of the application.

In this paper we present a new tool that is under development that tries to make the technology more accessible for programmers. Our tool is based on the same well-founded principles of existing work, but uses an intermediate verification language, Boogie, that improves the tool usability and scope in a number of ways. Using a general language for writing specifications makes specifications easier to write and improves expressiveness. Also, we leverage the language to add a library of CRDTs, which allows the programmer to solve conflicts without coordination. We discuss the features that we have already implemented and how they contribute to improve the technology.

CCS CONCEPTS

• Software and its engineering → Automated static analysis; **Consistency**; **• Computer systems organization → Cloud computing**;

KEYWORDS

Static verification; replication; integrity invariants

ACM Reference format:
Gonçalo Marcelino, Valter Balegas, and Carla Ferreira. 2017. Bringing Hybrid Consistency Closer to Programmers . In *Proceedings of PaPoC'17, Belgrade, Serbia, April 23, 2017,* 4 pages.
DOI: http://dx.doi.org/10.1145/3064889.3064896

1 INTRODUCTION

Replication is a fundamental technique for achieving better availability, scalability, and fault tolerance in contemporary storage systems. Many of these systems use a combination of *weak* and *strong* consistency models [? ? ?], coined as *hybrid consistency* in [?], to coordinate the execution of operations when the correctness of applications is at risk, and leverage the benefits of asynchronous execution when operations are safe.

To use *hybrid consistency*, programmers need to identify the application invariants that have to be maintained at all times, and instrument the application code to use coordination when necessary. Choosing the "right" coordination is a difficult task. For that, programmers have to reason about the concurrent effects of each operation in the application, in order to determine which operations require coordination, a complex process when dealing with large applications.

More recently a few static analysis tools have been proposed [? ? ?] that aid programmers to determine which operations have to be coordinated in order to maintain the application's correctness. These tools receive the specification of an application, and output the pairs of operations that might break the correctness of the application if executed concurrently. With that information, it is possible to derive sets of tokens, that can be associated with operations, to pin-point where in an application the coordination is required. However, existing tools have limitations that constitute a barrier for their adoption by practitioners: they use domain-specific languages not general enough for specifying the behavior of complex applications; they only support basic data types; and, identify conflicts at the grain of operations, leading to overly-conservative executions.

In this paper, we present a new tool that we are developing and discuss how it improves the usability of existing tools in a number of aspects. Our goal is to build a tool that can be used in practice to help programmers build correct applications using a *hybrid consistency* approach. As in [?] our tool automates the proof rule defined and proved sound in [?], ensuring that the coordination generated for a given application is correct.

In our tool the specifications are written in Boogie [?], a versatile intermediate verification language (used by Dafny [?] and VCC [?]). Boogie generates a set of verification conditions, from an input specification, and then uses a STM solver [?] to check those verification conditions. Opting for a verification language means giving the programmer the ability to easily specify more complex behaviors for operations, something lacking in previous works. We provide support for using complex data types, including those provided by the language or specified by the programmer. One particular case, is that we allow specifying conflict-free replicated data types (CRDTs) [?], which can be used to solve conflicting pairs of operations without coordination.

Furthermore we also propose a fine grained approach for operation coordination. Previous tools suggested the coordination of pairs of operations when their execution could potentially invalidate the invariant of the application under scrutiny. Our novel approach takes into account the parameters of these operations, advising operation coordination only for specific combinations of parameters, allowing the pairs of operations to execute concurrently in all remaining cases. This approach permits more concurrency, while still ensuring the preservation of the application's invariants.

Finally, when the tool detects a conflicting pair, it is capable of providing a counter-example that programmers can use to correct the application.

In the remaining of this paper we present each of these contributions in detail, and discuss the benefits that they provide to the tool.

2 SYSTEM MODEL

We consider a database system composed by a set of objects fully replicated over multiple data centers. An application is defined as a set of high-level operations and a set invariants that express well-formedness rules of the database state. Each operation is defined as a sequence of reads and updates, and has an associated precondition stating the conditions that have to be guaranteed for its safe execution. When an application submits an operation to the local replica, the precondition is checked on the local database state. If the precondition holds, the operation is executed locally, and its effects are propagate asynchronously to remote replicas. Otherwise, the operation has no effect. We assume that the propagation of operation effects respects causality.

We use a token system as the abstract coordination mechanism as defined in [?]. The token system consists of a set of tokens and a symmetric conflict relation over tokens. Each operation may have an associated set of tokens, ensuring that other operations with conflicting tokens cannot be executed concurrently and their execution has to be coordinated.

3 TOOL OVERVIEW

To apply our analysis the programmer has to annotate the application code with a specification. The specification describes the database state, data invariants, preconditions and effects of each operation. To illustrate the static analysis, we use as running example a distributed tournament management application. This application allows the following operations to be executed by its users: $addTournament(t)$ and $remTournament(t)$ register and remove tournament t, respectively, $addPlayer(p)$ registers player p, and $enroll(p,t)$ enrolls player p in tournament t. Additionally, the application is subject to the following integrity invariant: if player p is enrolled in tournament t, both player p and tournament t must be registered.

The input specification is then analysed in three distinct steps.

Safety analysis This first step ensures that none of the specified operations are able to invalidate the application's invariant by executing in standalone manner without any concurrency. This is done to validate the correction of the specification given as input. In our example, if operation $remTournament(t)$ did not have the precondition requiring that no player is enrolled in tournament t, the operation would fail the safety analysis.

Commutativity analysis This second step checks commutativity between all pairs of operations and outputs the subset of these pairs that are not commutative, as well as the sets of tokens needed to address this issue. This is achieved by executing all pairs of operations in both orders and, afterwards, verifying if the state after these executions is the same. In our example, operations $addTournament(t)$ and $remTournament(t)$ do not commute, while $addPlayer(p)$ and $addTournament(t)$ are commutative.

Stability analysis This last step provides the programmer with the set of pairs of operations that cannot execute concurrently, as they can break the application's invariant, as well as the sets of tokens needed to avoid their concurrent execution. This analysis verifies the stability of each operation precondition against all other operations effects. In practice it verifies if the effects of any operation invalidates the precondition of the operation under analysis. As an example, the precondition of $enroll(p,t)$ is not stable under concurrent execution of $remTournament(t)$, while the precondition of $addTournament(t)$ is stable under the effects of $enroll(p,t)$.

4 TOKEN GENERATION

As briefly explained, the tool generates a set of tokens that are used to prevent conflicting pairs of operations from executing concurrently. The tokens that our tool generates are based on the parameters of each operation. This allows more fine-grained concurrency control than previous approaches, which only identify conflicts per operation. More specifically, the tool tests different parameter values for each pair of operations, identifying in which cases different combinations of parameters might invalidate the invariants. We leverage the verification engine to detect efficiently the problematic combination of parameters. The output of the tool is a token system as described before, indicating the relations between conflicting parameters. Taking as an example the pair of operations $enroll(p,t)$ and $remTournament(t)$, the tool outputs that a token must be used if parameter t is the same in the two operations. A complete example is shown in the Appendix A.

Finally, as an alternative to automatically generating the tokens, the programmer can define her own token system. The tool is then able to determine if the provided token system ensures commutativity and stability of the application's operations, while advising the programmer to remove tokens that are not needed to assert these properties, if any. As reducing the number of tokens decreases coordination and leads to a more scalable application

5 CRDT SPECIFICATION

CRDTs are replicated data types that specify well-defined convergence rules that can be used to solve concurrency conflicts. Previous work [? ?] has demonstrated that these data types can be used to solve some conflicting pairs of operations without using coordination. However, the tools from those works do not provide support for specifying CRDTs.

Our tool allows specifying CRDTs and use them during the analysis process. Moreover, we have defined a library of generic CRDT types that can be used by the programmer. With this library the programmer has the choice between using the tokens or CRDTs, as a way to solve conflicting operations. In the Appendix B we provide a specification of the tournament application that uses a remove-wins CRDT set.

6 CONCLUSION

We have presented a tool to help programmers take full advantage of the hybrid consistency model. The tool is based on intermediate verification language, Boogie, which empowers programmers with the ability to write general code for specifying applications. We demonstrated the usefulness of the approach by adding support for CRDTs, as an alternative mechanism for solving conflicting pairs. Also, we have extended the existing algorithm for detecting conflicting pairs with support for parameter analysis, which allows more concurrency in applications.

Future work will be focused on giving programmers the ability to certify already existing source code using the specifications given to the tool, as this would allow the programmer to be sure that their specification reflects what is actually implemented. We plan to use available Boogie APIs for general purpose languages, as Java [?] and C [?], to support verification of real code. This is an important aspect for bridging the gap between specification and implementation.

We also plan to explore ways to reduce the annotation effort, since even with a restricted number of case studies some specification patterns emerge. These patterns could be explored to help programmers writing specifications, by automatically generating part of the annotations or at least by proving a set of *best practices* to be followed.

Acknowledgements. This work was partially supported by FCT-MCTES-PT NOVA LINCS project (UID/CEC/04516/2013), FCT/MCT SFRH/BD/87540/2012, EU FP7 SyncFree project (609551), and EU H2020 LightKone project (732505).

A TOURNAMENT EXAMPLE

This section starts by showing the input Boogie specification[1] for the tournament application. Although the specification code can be verbose a few patterns emerge through the specification code. These patterns could be helpful in reducing the programmer's annotation effort. One such pattern appears in the `ensures` clause that expresses the effects of operations. The update of a state set variable for a given argument is reflected in a `ensures` clause with a `forall` clause stating that the state variable for that argument is updated, while the remaining objects are left unchanged. This part of the clause for the objects not updated could be generated automatically by the tool, reducing the specification effort.

Input specification

```
type Tournament;
type Player;
var enrollment: [Player, Tournament] bool;
var tournaments: [Tournament] bool;
var players: [Player] bool;

function invariant() returns(bool)
{
   forall t: Tournament, p: Player ::
     enrollment[p,t] ==> tournaments[t] && players[p]
}

procedure addTournament(t1: Tournament)
modifies tournaments;
requires true;
ensures forall t: Tournament ::
  t == t1 ==> tournaments[t1] == true
  &&
  t != t1 ==> tournaments[t] == old(tournaments)[t]; { }
```

[1]For illustrative reasons we removed some parenthesis in clauses `requires` and `ensures`.

```
procedure remTournament(t1: Tournament)
modifies tournaments;
requires !exists p: Player :: enrollment[p, t1];
ensures forall t: Tournament ::
  t == t1 ==> tournaments[t1] == false
  &&
  t != t1 ==> tournaments[t] == old(tournaments)[t]; { }

// addPlayer and remPlayer can be similarly defined.

procedure enroll(p1: Player, t1: Tournament)
modifies enrollment;
requires players[p1] == true && tournaments[t1] == true;
ensures forall p: Player, t: Tournament ::
  p == p1 && t == t1 ==> enrollment[p1,t1] == true
  &&
  p != p1 || t != t1 ==> enrollment[p,t] == old(enrollment)[p,t]; { }
```

Safety analysis

With the previous specification the safety analysis does not report any errors. However, if we remove the precondition `players[p1] == true` from operation `enroll`, an invariant violation is reported. In this case, Boogie can build a counter-example model to help the programmer in correcting her specification.

```
enrollments -> [Player,Tournament]Bool
players -> [Player]Bool
tournaments -> [Tournament]Bool
p1 -> Player
t1 -> Tournament

Select_[Player,Tournament]Bool -> {
  [Player,Tournament]Bool p1 t1 -> true
  else -> true
}
Select_[Tournament]Bool -> {
  [Tournament]Bool  t1 -> true
  else -> true
}
Select_[Player]Bool -> {
  [Player]Bool p1 -> false
  else -> false
}
```

Stability analysis

```
Conflicting operations:
{ enroll(p, t), remPlayer(p) }
{ enroll(p, t), remTournament(t) }
```

Commutative analysis

Commutativity is only verified for pairs of operations that do not fail the stability analysis.

```
Non-commutative operations:
{ addTournament(t), remTournament(t) }
{ addPlayer(p), remPlayer(p) }
```

Generated tokens

```
enroll(p,t)      : { token_ep(p), token_et(t) }
remTournament(t) : { token_rt(t) }
addTournament(t) : { token_at(t) }
remPlayer(p)     : { token_rp(p) }
addPlayer(p)     : { token_ap(p) }

Conflict relation:
token_ep(p)  : token_rp(p)
token_rt(t)  : token_et(t), token_at(t)
token_at(t)  : token_rt(t)
```

```
token_rp(p)  : token_ep(p), token_ap(p)
token_et(t)  : token_rt(t)
token_ap(p)  : token_rp(p)
```

B AVOIDING COORDINATION WITH CRDTS

Generic remove-wins CRDT

```
type Selector;
type CRDTElement = <a>[a]bool;
type CRDT = [Selector]CRDTElement;
const unique add: Selector;
const unique remove: Selector;

axiom( forall s:Selector :: s == add || s == remove );

function CRDTAdd<a>(elem: a, set: CRDT, oldSet: CRDT) returns(bool)
{
  forall e:a :: e == elem ==> set[add][elem] == true
                &&
                e != elem ==> set[add][e] == oldSet[add][e]
  &&
  forall e:a :: set[remove][e] == oldSet[remove][e];
}

function CRDTRemove<a>(elem: a,
                       set: CRDT, oldSet: CRDT) returns(bool)
{
  forall e:a :: e == elem ==> set[remove][elem] == true
                &&
                e != elem ==> set[remove][e] == oldSet[remove][e]
  &&
  forall e:a :: set[add][e] == oldSet[add][e];
}

function CRDTInSet<a>(element: a, set: CRDT) returns(bool)
{
  set[add][element] && !set[remove][element];
}
```

Input specification with CRDT sets

```
type Tournament;
type Player;
var enrollment: [Player, Tournament] bool;
var tournaments: CRDT;
var players: CRDT;

function invariant(enrollment: [Player,Tournament] bool,
                   tournaments: CRDT,
                   players: CRDT) returns(bool)
{
   forall t: Tournament, p: Player ::
     enrollment[p,t] ==> CRDTInSet(p, players)
                         &&
                         CRDTInSet(t, tournaments)
}

procedure addTournament(t1: Tournament)
modifies tournaments;
requires true;
ensures CRDTAdd(t1, tournaments, old(tournaments)); { }

procedure remTournament(t1: Tournament)
modifies tournaments;
requires !exists p: Player :: enrollment[p, t1];
ensures CRDTRemove(t1, tournaments, old(tournaments)); { }

// addPlayer and remPlayer can be similarly defined.

procedure enroll(p1: Player, t1: Tournament)
modifies enrollment;
requires CRDTInSet(p1, players) && CRDTInSet(t1, tournaments);
procedure enroll(p1: Player, t1: Tournament)
ensures forall p: Player, t: Tournament ::
  p == p1 && t == t1 ==> enrollment[p1,t1] == true
  &&
  p != p1 || t != t1 ==> enrollment[p,t] == old(enrollment)[p,t]; { }
```

Stability analysis

```
Conflicting operations:
{ enroll(p, t), remPlayer(p) }
{ enroll(p, t), remTournament(t) }
```

Commutative analysis

All non-conflicting operations are commutative.

Generated tokens

```
enroll(p,t)      : { token_ep(p), token_et(t) }
remTournament(t) : { token_rt(t) }
addTournament(t) : { }
remPlayer(p)     : { token_rp(p) }
addPlayer(p)     : { }

Conflict relation:
token_ep(p)  : token_rp(p)
token_rt(t)  : token_et(t)
token_rp(p)  : token_ep(p)
token_et(t)  : token_rt(t)
```