Ownership-Based Order Reduction and Simulation in Shared-Memory Concurrent Computer Systems

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Dipl.-Ing. Christoph Baumann
baumann@wjpserver.cs.uni-saarland.de

Saarbrücken, im Januar 2014

Tag des Kolloquiums:	19. März 2014
Dekan:	Prof. Dr. Mark Groves
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Sebastian Hack
1. Gutachter:	Prof. Dr. Wolfgang J. Paul
2. Gutachter:	Prof. Dr. Bernhard Beckert
Akademischer Mitarbeiter:	Dr. Mikhail Kovalev

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im Februar 2014

"Education is an admirable thing.
But it is well to remember from time to time that
nothing that is worth knowing can be taught."
OSCAR WILDE, 1894

Acknowledgments

First and foremost I wish to express my gratitude to my advisor Prof. Dr. Wolfgang J. Paul for his patience and guidance in the development of this thesis. I also had the privilege to collaborate with him on many other topics and I am thankful for all that I have learned from him. Working at his chair has truly been a unique and valuable experience, that I would not want to have missed.

This is also notably due the friendly and productive atmosphere at the chair, for which I want to thank my past and present colleagues. I am especially indebted to Sabine Schmaltz who always was open to discussion and provided several helpful suggestions which led to the improvement of the theory presented in the sequel.

Life is concurrent and each day we take so many $\mathcal{IO}$ steps, therefore this thesis is dedicated to the main invariants in my life. In particular, I am grateful to my parents for their unconditional love and support and to my lifelong friend David for sticking by me through all these years. Moreover, I raise my glass to my 'companion in crime' Matthias, for being simply the "best guy ever".

Finally, I would like to acknowledge the mastership of Thomas Bruch, Christopher Hastings, and John Haughm, whose works have been an excellent source of motivation, inspiration, and consolation during the creation of this document.

Abstract

The highest level of confidence in the correct functionality of system software can be gained from a pervasive formal verification approach, where the high-level language application layer is connected to the gate-level hardware layer through a stack of semantic layers coupled by simulation theorems. While such semantic stacks exist for sequential systems, the foundational theory of semantic stacks for concurrent systems is still incomplete. This thesis contributes to close this gap.

First we prove a general order reduction theorem establishing a model where processes are executing blocks of steps, being only interleaved at selectable interleaving-points. An ownership-based memory access policy is imposed to prove commutativity properties for non-synchronizing steps, enabling the desired reordering. In contrast to existing work, we only assume properties on the order-reduced level, thus providing a complete abstraction.

We then apply sequential simulation theorems on top of the block schedules and prove a general simulation theorem between two abstract concurrent systems including the transfer of safety properties. Finally we instantiate our frameworks with a MIPS instruction set architecture, a macro assembler (MASM) semantics, and an intermediate language semantics for C. Applying the concurrent simulation theorem, we justify the concurrent semantics of MASM and C against their ISA implementation.

Kurzzusammenfassung

Das größte Vertrauen in die korrekte Funtionsweise von System-Software kann mit Hilfe durchdringender formaler Beweisverfahren erlangt werden, welche alle Abstraktionsebenen eines Computersystems durch Simulationstheoreme miteinander koppeln. Während solche Gerüste von Semantiken bereits für sequentielle Systeme entwickelt wurden, finden sich in der entsprechenden Theorie für nebenläufige Systeme noch Lücken, zur Schließung derer diese Arbeit beitragen soll.

Zunächst beweisen wir ein allgemeines Reduktionstheorem, das die möglichen Reihenfolgen, in der Prozesse Schritte machen, auf ein Modell beschränkt, in dem Blöcke von Schritten verschiedener Prozesse nacheinander ausgeführt werden. Mittels eines "Ownership"-basierten Speicherzugriffprotokolls beweisen wir Kommutativitätseigenschaften für lokale Schritte verschiedener Prozesse und ermöglichen so das Vertauschen dieser. Da unser Theorem nur Eigenschaften des reihenfolgereduzierten Systems annimmt ermöglicht es eine vollständige Abstraktion vom ursprünglichen Modell.

Auf die Blockausführung wenden wir sequentielle Simulationstheoreme an und beweisen ein allgemeines Simulationstheorem zwischen abstrakten nebenläufigen Systemen sowie den Transfer von Sicherheitseigenschaften. Wir instanziieren das Theorem mit einem MIPS-Instruktionssatz und Semantiken für Makroassembler und C. Dadurch rechtfertigen wir die nebenläufige Semantik der Programmiersprachen gegen ihre Maschinenimplementierung.

Contents

1 Introduction **1**
- 1.1 Overview . . . 2
- 1.2 Related Work . . . 6
- 1.3 Notation and Conventions . . . 10

2 The *Cosmos* Model **17**
- 2.1 Signature and Instantiation Parameters . . . 17
- 2.2 Configurations . . . 20
- 2.3 Restrictions on Instantiated Parameters . . . 21
- 2.4 Semantics . . . 22
- 2.5 Computations and Step Sequence Notation . . . 23
- 2.6 Ownership Policy . . . 25

3 Order Reduction **31**
- 3.1 Interleaving-Point Schedules . . . 31
- 3.2 Reordering into Interleaving-Point Schedules . . . 36
- 3.3 Auxilliary Definitions and Lemmas . . . 41
- 3.4 Commutativity of Local Steps . . . 51
- 3.5 Equivalent Reordering Preserves Safety . . . 54
- 3.6 Reduction Theorem and Proof . . . 55
- 3.7 Coarse $\mathcal{IO}$-Block Scheduling . . . 60

4 *Cosmos* Model Instantiations **63**
- 4.1 MIPS ISA . . . 63
  - 4.1.1 Instruction Set . . . 64
  - 4.1.2 Configuration . . . 68
  - 4.1.3 Semantics . . . 69
    - Auxiliary Definitions and Intermediate Results . . . 70
    - Regular Instruction Execution . . . 73
    - Interrupt Semantics . . . 74
    - Overall Transition Function . . . 77
  - 4.1.4 *Cosmos* Machine Instantiation . . . 77
- 4.2 Macro Assembly . . . 83
  - 4.2.1 MASM Syntax and Semantics . . . 83
    - Instructions . . . 83
    - Programs . . . 84

Configuration . . . 85
MIPS Assembler Calling Convention . . . 87
Transition Function . . . 89
4.2.2 MASM Assembler Consistency . . . 94
Stack Layout . . . 94
Assembler Information . . . 96
Simulation Relation . . . 96
Simulation Theorem . . . 99
4.2.3 Concurrent MASM . . . 102
4.3 C Intermediate Language . . . 105
4.3.1 C-IL Syntax and Semantics . . . 105
Environment Parameters . . . 105
Types . . . 106
Values . . . 108
Expressions . . . 110
Programs . . . 111
Configurations . . . 113
Transition Function . . . 120
4.3.2 C-IL Compiler Consistency . . . 124
Compilation and Stack Layout . . . 124
Compiler Consistency Points . . . 129
Compiler Consistency Relation . . . 131
Simulation Theorem . . . 135
4.3.3 Concurrent C-IL . . . 139

5 Simulation in Concurrent Systems 151
5.1 Block Machine Semantics . . . 151
5.1.1 Definition . . . 152
5.1.2 Reduction Theorem and Proof . . . 153
5.2 Generalized Sequential Simulation Theorems . . . 155
5.3 *Cosmos* Model Simulation . . . 160
5.3.1 Consistency Blocks and Complete Block Machine Computations 160
5.3.2 Requirements on Sequential Simulation Relations . . . 162
5.3.3 Simulation Theorem . . . 165
5.3.4 Auxiliary Lemmas . . . 167
5.3.5 Simulation Proof . . . 169
5.4 Applying the Order Reduction Theorem . . . 175
5.4.1 Corollaries of the Simulation Theorem . . . 176
5.4.2 Suitability and Reordering . . . 178
5.4.3 Order Reduction on Suitable Schedules . . . 179
5.5 Property Transfer and Complete Block Simulation . . . 180
5.5.1 Simulated *Cosmos* machine Properties . . . 180
5.5.2 Property Transfer . . . 182

5.6 Instantiations 184
5.6.1 Concurrent MASM Assembler Consistency 184
Sequential Simulation Framework 184
Shared Invariant and Concurrent Simulation Assumptions 187
Proving Safety Transfer 188
5.6.2 Concurrent C-IL Compiler Consistency 190
Sequential Simulation Framework 190
Shared Invariant and Concurrent Simulation Assumptions 192
Proving Safety Transfer 193

6 Conclusion **195**
6.1 Discussion 196
6.2 Application 201
6.3 Future Work 202
6.3.1 Device Step Reordering 202
6.3.2 Store Buffer Reduction 203
6.3.3 Interrupt Handler Reordering 203

1 Introduction

The formal verification of concurrent computer systems has come a long way from the development of the first methodologies in the 70s. Today there are expressive program logics and powerful verification tools allowing to prove the functional correctness of complex concurrent algorithms and real-world system software. What many of the verification approaches have in common, is that they are arguing about programs written in some abstract high-level language containing primitives for synchronous interaction between processes. For such languages a formal semantics is usually easy to define compared to real-world programming languages that system software is written in, e.g., C mixed with assembly portions. The program logic employed for verification is then proven sound against the semantics of the programming language, strengthening the confidence in the correctness of programs that were verified to be correct.

However, there is a gap in this kind of reasoning. Even when considering more detailed, realistic programming semantics, the fact stands that computers naturally do not execute high-level language programs. They do not even execute assembly programs. They execute machine code. So the question is: Why has the abstract concurrent semantics employed in program verification anything to do with the semantics of the translation to machine code being executed on a multicore processor with shared memory? In many cases it is silently assumed that the target programming language of a verification method correctly abstracts from some real-world implementation, relying on the correctness of compilers and assemblers used for the refinement of the program to the hardware level. For the sequential setting indeed correctness proofs for compilers and assemblers have been conducted, allowing to transfer verified properties of the abstract program down to the assembly language and instruction set architecture layers. In general, for a pervasive verification approach, one needs to built stacks of semantics where adjacent levels of abstraction are coupled via sequential simulation theorems. One big advantage of this apporach is that, by proving the simulation theorems, one uncovers *all* the required software conditions, that make the simulation go through, justifying the abstraction itself. Nevertheless, for the construction of concurrent semantic stacks a complete formal theory is still missing, leaving a wide gap in the soundness proofs of verification methods and tools for concurrent systems.

A straight-forward approach for establishing a concurrent semantics stack is to employ the simulation theorems from a sequential semantic stack in the concurrent setting. However, this is impeded by the *interleaving model* of concurrency where processes take steps one after another in an arbitrary order. Thus in general there is no sequence of steps by a single process on which a sequential simulation theorem could be applied. Nevertheless the order of steps by different processors should only matter for steps

that are synchronizing the process with its environment, e.g., through shared memory accesses. In fact, it is a well-known folklore theorem that between synchronizing steps the interleaving of local operations is irrelevant. We use this fact in order to prove an order reduction theorem which abstracts from arbitrarily-interleaved computations of the concurrent system and allows to reason exclusively about schedules where processes execute blocks of steps on which sequential simulation theorems like compiler correctness can be applied.

In a second step we then compose the sequential simulations on blocks of steps into a system wide concurrent simulation theorem. This approach has two main hypotheses. The first one is that processes are well-behaved in the sense that they do not break the simulation relations holding for others. The second is that the simulation theorems allow the transfer of the safety conditions that enable the application of the order reduction theorem on the lowest abstraction layer. These safety conditions require mainly the absence of memory races between processes, enabling the commutativity of steps. We use an explicit ownership model and define an ownership-based memory access policy that prevents memory races.

Overall, this document provides (1) a framework to model concurrent systems with shared memory at any level of abstraction (2) a concurrent simulation theorem to relate different abstraction layers (3) a general order reduction theorem for concurrent systems that enables applying the simulation theorem (4) a complete correctness proof of our approach – including the aforementioned theorems – in paper and pencil mathematics, and (5) instantiations of our framework justifying the concurrent semantics of macro assembly and intermediate-language C programs that are implemented on a MIPS multiprocessor.

1.1 Overview

As motivated above we want to look at asynchronous concurrent systems on different layers of abstraction and connect these layers formally using simulation relations, or refinement mappings respectively. This requires defining formal semantics for every layer. In order to have a uniform representation of these semantic levels, we introduce a generic framework that we call the *Concurrent System with Shared Memory and Ownership*, abbreviated by the acronym *Cosmos*. A *Cosmos* model consists essentially of a number of infinite state machines called *computation units*, which can be instantiated individually, and a *shared memory* containing the common system resources. In addition there are specification components representing the dynamic *ownership* state.

Following O'Hearn's ownership hypothesis [O'H04], that the memory of race-free programs can be divided into disjoint portions that are owned exclusively by the different computation units, we use a simple ownership model to keep track which memory addresses are owned by a particular machine in the system. Additionally some addresses are shared by all participating machines and may only be accessed in a synchronized fashion, e.g., by using interlocked atomic memory operations. We specify ownership invariants and a memory access policy based on ownership that provably

guarantees the absence of memory races (memory safety) if all computation units adhere to it. Moreover we can use ownership later on to prove commutativity of certain steps by different units. Thus it is a cornerstone principle of our approach. We require that safe operations preserve ownership invariants and perform only safe memory accesses wrt. the ownership model, i.e., they must obey the ownership policy. This policy demands, e.g., that one machine does not write addresses owned by other machines or that shared memory may only be accessed by designated synchronizing steps, so-called $\mathcal{IO}$ *steps*, where $\mathcal{IO}$ stands for input/output. Every instantiation of a *Cosmos* model machine must specify which of its state transitions are considered $\mathcal{IO}$.[1] Intuitively an $\mathcal{IO}$ step represents a synchronization operation, and thus a communication between the computation units. Usually it is implemented via an access to shared memory, however different instantiations may have different ways of synchronization.

Now we can instantiate *Cosmos* machines describing the same system at different levels of detail. For instance, we might look at a computer system were C programs are executed concurrently on multiple processor cores. In the real world these programs are compiled to a certain multicore instruction set architecture (ISA) which we can also represent by a *Cosmos* model (possibly a reduced version under certain software conditions). Each program is compiled separately and there exists a sequential simulation relation as well as a simulation theorem that links C and ISA execution on one processor. It would be desirable to compose the sequential simulation theorems to one global simulation theorem spanning all processors in the *Cosmos* model. Naturally, this would base on the hypothesis that for every particular unit its sequential simulation relation is preserved by executions of the overall system. However we can not prove such a theorem for arbitrary interleavings of machine steps because a simulation relation need not generally hold for all states of an execution on the abstract level and usually only holds for a few steps on the concrete level where the steps are refined[2]. In fact we need to assume schedules on the low level where units are executing blocks of steps from one consistency point (a configuration that is consistent to/simulates an abstract one) until the next one. Furthermore we cannot just combine arbitrary programs in a concurrent fashion. There is a number of requirements on the nature of the simulation theorems under consideration and the notion of the shared resources on both levels. We list all the necessary assumptions including memory safety on the abstract level and prove the global *Cosmos* model simulation theorem based on the correctness of the local simulations which have to be verified individually. One important assumption is that ownership-safe computations on the abstract level can be implemented by ownership safe computations on the concrete level of the simulation theorem. This property allows the top-down transfer of ownership-safety in a semantic stack and needs to be proven for every sequential simulation theorem involved. The assumption of block scheduling is justified by an order reduction theorem.

In order reduction we reduce the number of possible interleavings of different pro-

[1] The individual definition of $\mathcal{IO}$ must allow to verify the hypotheses of our order reduction theorem for a given program, most prominently the memory safety conditions.

[2] The only exception to this is one-step simulations where one abstact step maps to a single concrete one.

cessor steps. The core argument to enable reduction is that the effect of a safe concurrent system execution does only partly rely on the scheduling of units. In fact it relies only on the external input sequence and the schedule of $\mathcal{IO}$ *steps*. All non-$\mathcal{IO}$ actions can be reordered arbitrarily as long as program order is maintained for each computation unit. The reordering preserves the effect of the original trace because non-$\mathcal{IO}$ steps only modify local information. This can be enforced by the ownership policy and is the main lemma of the proof. Instead of non-$\mathcal{IO}$ operations we also speak of local steps and we call the configuration from which an $\mathcal{IO}$ step origins $\mathcal{IO}$ point.

For the reduction theorem we assume a definition of *interleaving-points* ($\mathcal{IP}$) that is provided by the verification engineer for a given instantiation and we reduce arbitrary interleavings to so-called $\mathcal{IP}$ *block schedules*, which are interleavings of blocks of steps of the same computation unit. Each $\mathcal{IP}$ block starts in an interleaving-point and contains at most one $\mathcal{IO}$ step. If we focus on the execution of a single unit in the computation we see that its sequential execution is only interrupted at the interleaving-points and $\mathcal{IP}$ blocks of other units are interleaved. The reduction theorem requires that all $\mathcal{IP}$ block schedules are proven to obey the ownership policy ensuring the absence of memory races. On the other hand it guarantees that the ownership policy is obeyed by arbitrary schedules and for each of them we can find a consistent $\mathcal{IP}$ block schedule. Additionally we prove that the reduction allows the transfer of arbitrary *safety properties*[3] from the order-reduced to the arbitrarily interleaved level. We do not consider liveness here.

As the interleaving-point schedules can be chosen arbitrarily[4], we can actually apply the order reduction theorem in two different scenarios. Our main target is the construction of a concurrent semantic stack, thus we need to choose the interleaving-points in a way such that we get $\mathcal{IP}$ blocks on which the sequential simulation theorems can be applied. To this end, in the simulation scenario, we choose the interleaving-points to be the consistency points. If we consider the top level of the simulation stack where the actual program verification is performed, a verification tool usually works more efficiently the less interleavings of steps by other units one has to consider. Thus, in the verification scenario we only interleave the actions of other computation units before $\mathcal{IO}$ steps, i.e., we choose the interleaving-points to be the starting points of $\mathcal{IO}$ steps. See Fig. 1 and 2 for an illustration of reordering for the two different choices of interleaving-points described above.

Thus we establish an abstract framework for order reduction and simulation in concurrent systems. We show its applicability by a number of instantiations. The *Cosmos* model is instantiated with a simplified MIPS multiprocessor, semantics for macro assembly programs (MASM), and semantics for an intermediate language of C (C-IL). We plug these systems into our simulation theory and show how the concurrent C and macro assembly machines can be proven to be simulated by the MIPS machine.

[3] Intuitively a safety property is an invariant on the execution sequence of a program, stating that something undesirable does not happen [Lam77]. Thus safety properties describe properties like the absence of run-time errors, race-freedom, or partial correctness. More precisely, "a safety property is one that holds for an infinite sequence if it holds for every finite prefix of it" [ALM84], hence it only restricts finite behaviour, as opposed to liveness properties which only restrict infinite behaviour [AS85].

[4] There may be at most one $\mathcal{IO}$ step between two interleaving-points of the same unit (cf. Sect. 3.1).

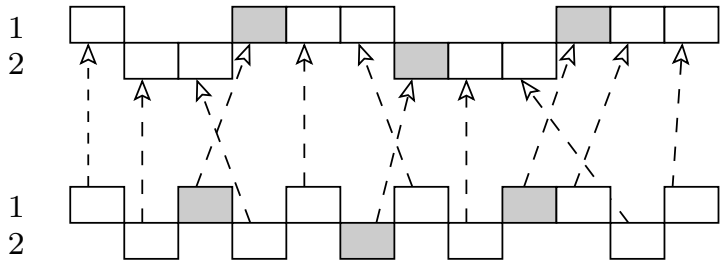

Figure 1: Illustration of reordering such that steps are only interleaved before $\mathcal{IO}$ steps. Filled boxes mark $\mathcal{IO}$ steps, empty boxes are local steps of the correspronding computation unit. Arrows display the permutation of steps.

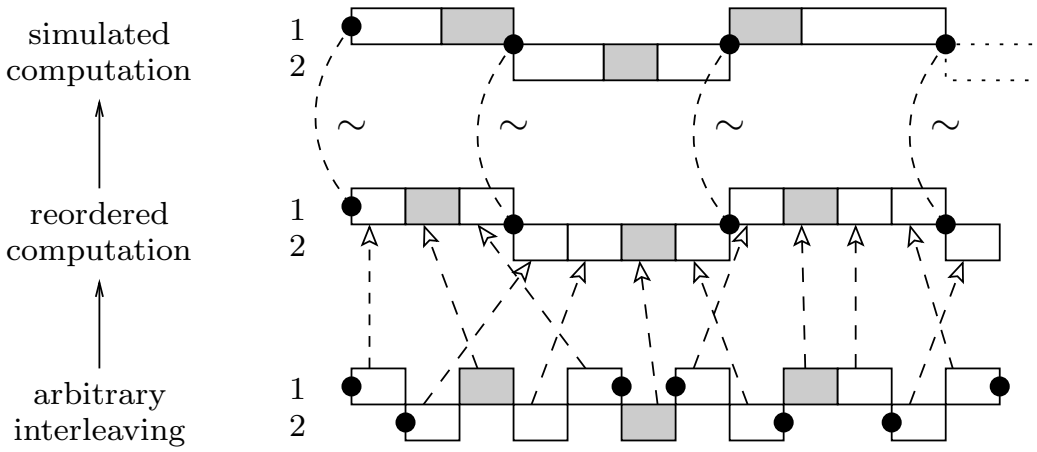

Figure 2: Illustration of reordering such that steps are only interleaved in compiler consistency points, allowing for simulation of a more abstract computation. Here dashed lines annotated with $\sim$ represent the compiler consistency relation holding in consistency points. Unit 2 has not yet reached a consistency point by the end of the computation, therefore the step it is simulating (suggested by dotted lines) is not yet considered on the abstract level.

The subsequent sections are structured as follows. In Chap. 2 we introduce the *Cosmos* model and the ownership policy. We procede in Chap. 3 to define the order reduction theorem and prove it. Chapter 4 contains instantiation examples of the *Cosmos* model with semantics for the MIPS instruction set architecture, MASM, and C-IL. In Chap. 5 we first set up a framework to define sequential simulation theorems in a unified way. We then list the requirements needed to combine the sequential simulations on separate units into a concurrent one. Finally we prove the concurrent simulation theorem and present instantiations of the framework with simulations between MIPS and MASM, as well as MIPS and C, discharging the aforementioned requirements on the sequential simulation theorems. We conclude this thesis by a discussion of our approach and an outlook on future work.

1.2 Related Work

The first formal approaches for the specification and verification of sequential programs date back to the 60s [Flo67, Hoa69]. In the subsequent decades myriads of different tools and languages have been developed to tackle the problem in a more efficient way and to apply techniques to more and more complex systems. The most challenging targets for formal methods may be operating systems, and recent history has seen several attempts on the verification of sequential OS microkernels [Kle09]. The approach of the Verisoft project [AHL+09] however comes probably closest to our vision of modelling computer systems on various abstraction levels that are linked by simulation theorems, thus enabling *pervasive* verification. There a semantics stack was developed spanning abtraction layers from the gate level hardware description up to the user application view. Nevertheless only a sequential, academic system was considered. The succeeding Verisoft XT project aimed at transferring this approach to the real world and developed tools and methods for the formal verification of industrial concurrent operating systems and hypervisors [CAB+09, LS09, BBBB09]. The theory presented in this report was conceived in an effort to justify the specification and verification approach used in the Verisoft XT project, were the automated verifier VCC [CDH+09] was employed to prove code correctness on the concurrent C level.

The VCC tool is just another step in the long history of specification and verification methods for concurrent systems [Lam93]. As early as 1975 Ashcroft [Ash75] and Owicki/Gries [OG76] extended Floyd's and Hoare's methods to the concurrent case. The approach was based on assertions that were added to programs in order to show the preservation of global invariants. Instead of using programs to specify other programs Lamport suggested to use state machines and mathematical formulas as a universal specification language. His Temporal Logic of Actions (TLA) [Lam94] allows to define safety and liveness properties of programs and algorithms in a uniform and concise way. Systems can be composed by disjunction of TLA specifications [AL95] and specified at different levels of abstraction. In case a refinement mapping exists between two layers it can be shown that one specification implements the other by showing an implication in TLA [AL91]. However, this simulation approach seems only to be suitable for

the program refinement or simple simulation theorems where the concrete specification makes steps in parallel with one (posibly stuttering) abstract step. For general simulation theorems where m concrete steps simulate n abstract ones we can not use the result directly. We would need to merge the n abstract simulated steps into an atomic step, however in a concurrent system this requires order reduction.

In 1978 Hoare introduced an algebraic method to model communicating sequential processes [Hoa78]. Subsequently the language he proposed evolved into the well-known CSP process algebra [Hoa85] and a whole community following his approach was formed. Several other process algebras followed in the wake of CSP, most prominently Milner's π-calculus. However this methodology seems only appropriate to model systems at a very abstract level, e.g., to specify algorithms and protocols. Modelling complex systems by algebraic structures and their transformations is at best messy, if not infeasible.

Another approach to concurrent system specification are I/O Automata as introduced by Lynch and Tuttle [LT87]. I/O Automata are basically infinite state machines which are characterized by actions they can perform. Internal actions are distinguished from externally visible input and output actions that are employed for communication between automata. Lynch uses these automata to model sequential, parallel and asynchronously concurrent shared memory systems and algorithms [Lyn96]. It is possible to compose several automata into a bigger one given certain composability requirements on their actions. Moreover it was shown how to prove simulation theorems between automata. However it was not treated how automata working on shared variables should be composed (an overall automaton containing all sub-automata was assumed) nor how simulation is maintained by composition of I/O automata. Finally the need to list all actions of an automaton and divide them into input, output and internal ones appears to be feasible for small-scale systems and algorithms. Modelling a realistic system like a modern multi-core instruction set architecture as an I/O automaton, in contrast, seems to be a rather tedious task. Nevertheless we feel that using composable state machines is the right methodology in order to obtain a formally linked semantics stack. In this way we are inspired by the work of Lynch and Lamport. Gurevich also followed the idea of modelling systems as *Abstract State Machines* (ASMs) [Gur00, BG03] and a programming language exists to specify and execute such models [GRS05]. However the ASM approach does not support asynchronous concurrency [Gur04]. As to the best of our knowledge there is no prominent formalism in literature general and convenient enough to serve our purposes we define our own framework for modelling concurrent systems using well-known concepts like automata theory and simulation relations. Here we also draw on the experience from the Verisoft project where these methods were applied successfully.

Concerning order reduction a multitude of related work exists. It appears that Lipton was the first to describe how several steps of an operation could be aggregated into a monolithic atomic action [Lip75]. Subsequently, Doeppner [Doe77], and Lamport and Schneider [LS89] showed that order reduction allows for the transfer of arbitrary safety properties, i.e., that if a safety property holds for all order-reduced interleavings of a

program execution then it also holds on all arbitrary interleavings. The main prerequisite for the reduction is that the reduced operation contains only one access to shared memory. The approaches described above had in common that a particular program is examined where the sub-steps of a specific operation are reordered to form a block of steps which is executed atomicly. To this end it was required for the non-critical steps in the reduced operation to commute with other concurrent program steps such that their combined effect would be preserved.

Later Cohen and Lamport generalized the reduction argument in Lamport's Temporal Logic of Actions (TLA) [CL98], showing also the preservation of liveness properties, which we do not treat in this report. It is possible, though non-trivial, to instantiate the TLA order reduction theorem in a way that would give us the desired block structure for arbitrary program executions. Thus our reduction theorem is a special case of the ones given in [LS89] and [CL98], using an ownership policy to justify the commutativity of steps. However, since the TLA reduction theorem requires *on the unreduced level* S that local steps of one process commute with steps of other processes, we cannot apply the TLA reduction theorem directly to achieve what we need. While the ownership policy enables us to show the commutativity of steps in a straight-forward way, we do not assume ownership-safety on S – we only assume ownership-safety on the order-reduced specification S^R. Thus, we need to transfer ownership-safety from S^R to S explicitly. In contrast to the existing theories described above, we also do not formalize order reduction in a way such that reordered steps are merged into single atomic actions; the effect of order reduction is expressed in terms of program schedules given by input sequences. We aim for a scheduling, where computation steps by the same process are reordered into blocks that start in the selected interleaving-points. We generalize the reordering to an equivalence relation between arbitrary schedules and corresponding block schedules. The transfer of safety properties works similar to [LS89] and [CL98].

Besides the classic theorems there is more recent work on order reduction. Several specific approaches have been proposed to employ reduction in model checking in order to tackle the problem of state explosion [FQ04, FFQ05, FG05]. In the spirit of reducing the number of interleavings of concurrent programs, Brookes [Bro06] presents a denotational semantics for high-level-language[5] shared-variable concurrent programs. In the semantics the possible computations of a program are represented by a set of sequences of state transitions (*footsteps*) which are interleaved at most at *synchronization points*, where resources are acquired. Sequential and concurrent program steps are merged into joint footsteps in case there are no data races between them, thereby pruning redundant interleavings of transitions which result in the same system state. However the semantics does not seem to support the simulation proofs between different semantic layers we have in mind.[6] By merging steps of different processes the

[5] A heap memory manager as well as synchronization primitives for acquiring and releasing resources are assumed.

[6] The sequential simulation theorems we consider are formulated in a way that if the implementation on process p makes m steps out of configuration c, there exist n steps out of the abstract configuration d

notion of separate programs that are executed concurrently is lost, because for one footstep it is not visible which process(es) are executed for how many steps under what sequence of external inputs. Also only the parts of the state which were read and written are recorded in the footsteps. Therefore the representative transition sequences given by the footstep semantics are not suitable to apply the sequential simulation theorems forming our concurrent model stack.

Also the formal verification tool VCC [CDH+09] used in the Verisoft XT project relies on order reduction. Threads are verified as if they run sequentially and concurrent actions of other threads are only interleaved before shared memory accesses. A proof sketch justifying this kind of "coarse scheduling" in the VCC tool was presented in [CMST09], however it was not shown how verified safety properties on coarse schedules can be transferred to arbitrarily interleaved schedules. Nevertheless the ideas of a general order reduction theorem from [CMST09] were the starting point of this work.

Applying a memory access policy in order to enforce separation between different actors in a concurrent system is also a common approach, e.g., Concurrent Separation Logic [O'H04] induces a notion of ownership and its transfer. The shared memory is divided into fixed partitions where each partition is uniquely identified by a *resource* name and accompanied by an individual resource invariant asserting the well-formedness of the shared data. Concurrent processes can acquire and release exclusive ownership of resources and their protected portion of shared memory by using predefined synchronization primitives. For every process there is a safety policy demanding that only owned portions of shared memory may be accessed and that the resource invariants must hold before a resource may be released. If all processes obey this policy, their parallel execution can be decomposed into local computations where each process executes without interference by other processes [Bro04]. This result facilitates the *local reasoning* in separation logic about separate processes of concurrent programs where the changes of other processes on the unowned portion of shared memory are incorporated in the resource-acquiring step as a non-deterministic transition that is obeying the resource invariants.

The Verified Software Toolchain [App11] applies separation-logic-based verification on the level of abstract program semantics of concurrent systems. Their approach is justified by adding a model of access permissions to the verified sequential compiler CompCert's memory model [LABS12], however there do not seem to be any published results yet presenting how the sequential compiler correctness translates to the concurrent case. Finally, it should be also noted that commutativity arguments like the ones we use in the proof of our reduction theorem are closely related to the notion of non-interference in information flow theory [GM82, Rus92]. Simply put, it is argued that an action does not interfere with a certain domain if it can be pruned from a computation without changing results visible in that domain. If this is the case it can be placed elsewhere in the computation, yielding commutativity.

such that the simulation relation is maintained, i.e., $c \sim d \implies c' \sim d'$.

1.3 Notation and Conventions

In the scope of this document we use the following notation and conventions.

Definitions and Proofs

Important Definitions in the following thesis are numbered and printed in italic font. Shorter function and type definitions are marked by the term def that is written over the identity symbol. We implicitly assume free variables occurring in definitions to be universally quantified. Moreover in case a definition is a conjunction of several statements we allow to omit the $\wedge$ symbol between the conjuncts and instead display them as a numbered list with lower-case Roman numerals. For example instead of

$$P(x) \stackrel{def}{\equiv} A(x) \wedge B(x) \wedge C(x)$$

we can write:

$$P(x) \stackrel{def}{\equiv} \begin{array}{ll} \textit{(i)} & A(x) \\ \textit{(ii)} & B(x) \\ \textit{(iii)} & C(x) \end{array}$$

When applying lemmas in proofs it is sometimes not immediately clear how the quantified variables map to the variables in the specific scenario we are considering. In order to clarify how a lemma is instantiated, we use the notation $v := v'$ denoting that every occurrence of variable v in the lemma should be replaced by instance v'.

Set Notation

Generally, we use standard set theory, however we extend it by a few shorthand definition. The disjoint union of sets is denoted by $\uplus$. Let $A_1, \ldots A_n, B$ be sets, then:

$$B = A_1 \uplus \cdots \uplus A_n \stackrel{def}{\equiv} B = \bigcup_{i=1}^{n} A_i \ \wedge \ \forall i, j.\, i \neq j \Longrightarrow A_i \cap A_j = \emptyset$$

To express that two sets are equal for some element α, i.e. that α is either contained or not contained in *both* sets, we choose the following notation

$$A =_\alpha A' \stackrel{def}{\equiv} \alpha \in A \Longleftrightarrow \alpha \in A'$$

If both sets are subset of some superset B, i.e., $A, A' \subseteq B$, we can easily show the property, that if A and A' agree on all elements of B, they must be equal.

$$(\forall \alpha \in B.\, A =_\alpha A') \Longleftrightarrow A = A'$$

For any non-empty set A we use the Hilbert choice operator ϵ to select one of its elements. In particular for singleton sets we have $\epsilon\{a\} = a$. Finally we denote the cardinality of a finite set A by $\#A$.

Types, Records, and Functions

Types are usually identified by blackboard bold, e.g. $\mathbb{F}$ in case their names consist of a single letter. The natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$ do *not* contain zero. $\mathbb{N}_i \subset \mathbb{N}$ with $i > 0$ and $\#\mathbb{N}_i = i$ defines the set of the i smallest natural numbers.

$$\mathbb{N}_i = \{1, \ldots, i\}$$

In contrast, we use $\mathbb{N}_0$ to denote the natural numbers including zero, i.e. $\mathbb{N}_0 \stackrel{def}{=} \mathbb{N} \cup \{0\}$.

We treat record types as n-tuples where each component can have a different type. For updating component $r.x$ of a record r with a new value v we can use the notation $r' = r[x := v]$ which implies that r' is the update record where $r'.x = v$ and for all other components $c \neq x$ we have $r'.c = r.c$.

In general, new types are defined using the standard mathematical notation where "$\times$" creates tuples and "$\to$" creates total functions. Partial functions are identified by a "$\rightharpoonup$" arrow in their type signature. For functions f mapping elements from domain A to image B, we can obtain the domain set of f by the function $\mathrm{dom}(f) = A$. In particular for partial functions $\mathrm{dom}(f)$ returns the set of arguments for which f is defined. For $f : A \to B$ and a set $X \subseteq A$ we can obtain the restriction of f to set X with the following notation.

$$f|_X \stackrel{def}{=} \lambda x \in X.\ f(x)$$

We can update functions $f : A \to B$ at entries $i \in A$ with a new value $v \in B$ as follows.

$$f[i \mapsto v] \stackrel{def}{=} \lambda j \in A.\ \begin{cases} v & : \quad j = i \\ f(j) & : \quad \text{otherwise} \end{cases}$$

The composition of partial functions $f, f' : A \rightharpoonup B$ with disjoint domains is denoted by $f \uplus f'$, where $\mathrm{dom}(f \uplus f') = \mathrm{dom}(f) \cup \mathrm{dom}(f')$.

$$f \uplus f' \stackrel{def}{=} \lambda a \in \mathrm{dom}(f) \cup \mathrm{dom}(f').\ \begin{cases} f(a) & : \quad a \in \mathrm{dom}(f) \\ f'(a) & : \quad a \in \mathrm{dom}(f') \end{cases}$$

By adding "$\bot$" to the image set in order to denote undefined results, any partial function $f : A \rightharpoonup B$ can be turned into a total function $f : A \to B \cup \{\bot\}$, given that $\bot \notin B$. Instead of $a \notin \mathrm{dom}(f)$ we can therefore also write $f(a) = \bot$ synonymously.

Propositional Logic

Logical propositions contain conjunction $\wedge$, disjunction $\vee$, negation, implication $\Longrightarrow$, equivalence $\Longleftrightarrow$ and brackets. For negation literature knows several symbols.

$$/x \quad \stackrel{def}{=} \quad \sim x \quad \stackrel{def}{=} \quad \neg x \quad \stackrel{def}{=} \quad \overline{x}$$

Here we will use mostly $/x$ and sometimes $\overline{x}$ where it saves brackets. The priority order $\prec$ of logical operators used in this document is defined below from weakest to strongest binding.

$$\iff \prec \implies \prec \vee \prec \wedge \prec /$$

Quantifiers $\forall$ (universal) and $\exists$ (existential) have the weakest binding. In order to limit their scope we need to use surrounding parantheses.

Sequences

Moreover in this thesis we will deal excessively with computation sequences. By $a \in \mathbb{A}^*$ we represent an arbitrarily long but finite sequence of elements with type $\mathbb{A}$. We denote the elements of the sequence by adding a subscript index starting with 1. A sequence of length n can thus be defined by listing its elements.

$$a = a_1 a_2 a_3 \cdots a_n$$

Let ε be the empty sequence, then we introduce the length of finite sequences.

$$|\varepsilon| = 0 \qquad \forall \mathbb{A}, x \in \mathbb{A}, a' \in \mathbb{A}^*.\ |xa'| = |a'| + 1$$

For manipulating sequences we define the function *pop* which removes the first i members of a sequence.

$$pop(a,i) \stackrel{def}{\equiv} \begin{cases} a & : \quad i = 0 \\ pop(a', i-1) & : \quad i > 0 \wedge \exists x \in \mathbb{A}.\ a = xa' \\ \varepsilon & : \quad \text{otherwise} \end{cases}$$

Function tl yields the remainder after removing the head $hd(a) \stackrel{def}{\equiv} a_1$ of a sequence.

$$tl(a) \stackrel{def}{\equiv} pop(a, 1)$$

Concatenating finite sequences is straight forward. To improve legibility we allow to use an explicit concatenation operator $\circ$.

$$a = a\varepsilon = \varepsilon a \qquad ab = a_1 \cdots a_n b_1 \cdots b_n \qquad a \circ b \stackrel{def}{\equiv} ab$$

We can select finite subsequences of a sequence via interval notation. First of all for $a \leq b$ we introduce the following integer intervals.

$$\begin{array}{lcllcl} [a:b] & \stackrel{def}{\equiv} & \{a, a+1, \ldots, b\} & [a:b) & \stackrel{def}{\equiv} & [a:b-1] \\ (a:b] & \stackrel{def}{\equiv} & [a+1:b] & (a:b) & \stackrel{def}{\equiv} & [a+1:b-1] \end{array}$$

Now subsequences of x are easily defined recursively. Let $0 \leq a \leq b$, then:

$$x[a:b] \stackrel{def}{\equiv} \begin{cases} x_a & : \quad a = b \\ x_a x[a+1:b] & : \quad \text{otherwise} \end{cases}$$

For open intervals, e.g., $x[a:b)$, the equivalent closed interval from above shall be used. We overload the $\in$-operator from set theory to denote that an element $x \in \mathbb{A}$ is part of a sequence $a : \mathbb{A}^*$.

$$x \in a \stackrel{def}{\equiv} \exists i \in \mathbb{N}.\ a_i = x$$

Computations

Computations are sequences of configurations from a state space S. They rely on transition functions $\delta : S \times I \to S$ which transform the state using inputs from domain I. A computation $s \in S^*$ for input sequence $in \in I^*$ is obtained by applying the transition function using the appropriate input. To denote state transitions we use the following arrow notation for $i, n \in \mathbb{N}$.

$$s_i \longrightarrow^n_{\delta,in} s_{i+n} \stackrel{def}{\equiv} \begin{cases} s_{i+1} = \delta(s_i, in_i) \wedge s_{i+1} \longrightarrow^{n-1}_{\delta,in} s_{i+n} & : \quad n > 0 \\ 1 & : \quad n = 0 \end{cases}$$

It can be generalized for states $s, s' \in S$ omitting the index i.

$$s \longrightarrow^n_{\delta,in} s' \stackrel{def}{\equiv} \exists a \in \mathbb{S}^*.\ a_1 \longrightarrow^n_{\delta,in} a_{n+1} \wedge s = a_0 \wedge s' = a_{n+1}$$

A special versions of this is $s \longrightarrow_{\delta,in} s' \stackrel{def}{\equiv} s \longrightarrow^1_{\delta,in} s'$. For arbitrarily long computations we write:

$$s \longrightarrow^*_{\delta,in} s' \stackrel{def}{\equiv} \exists n.\ s \longrightarrow^n_{\delta,in} s'$$

If δ is a transition function without any input besides the pre-state of the transition, we can define the same notations as above by simply omitting the input.

Context-free Grammars

We assume the reader to be familiar with the concept of context-free grammars $G = (T, N, S, P)$ and the way they are producing languages $L(G)$. Here T denotes the terminal alphabet, N with $N \cap T = \emptyset$ the non-terminals, $S \in \mathbb{N}$ the starting symbol and $P \subseteq N \times (N \cup T)^*$ the production system. In one place we will use a context-free grammar in order to define the syntax of instruction representation for a MIPS instruction set architecture model.

If $n \in \mathbb{N}$ is a non-terminal and $a \in (N \cup T)^*$ a sequence of terminals and non-terminals, we write $n \longrightarrow a$ as a shorthand for $(n, a) \in P$. Several productions for the same non-terminal are seperated by "|", e.g., $n \longrightarrow a \mid b$ stands for $\{(n, a), (n, b)\} \subset P$. For finite sets $A \subsetneq T^*$ of words over the terminal alphabet and a non-terminal $n \in N$ we allow the following shorthand when defining the production system.

$$n \longrightarrow A \iff \forall a \in A.\ (n, a) \in P$$

Thus, for instance, a production rule $b \longrightarrow \mathbb{B}^{32}$ means that b can be replaced by any 32-bit string. In our grammars the terminal "␣" denotes a space. However in productions we do not use the symbol, but normal spaces for legibility. Moreover we use the notation $n \longrightarrow_G^* w$ to say that terminal word $w \in T^*$ can be derived from non-terminal $n \in N$ by a finite number of production steps wrt. grammar G.

Binary Arithmetic

When introducing our MIPS ISA instantiation of the *Cosmos* model we will need to argue about arithmetics on bit strings. Bit strings are finite sequences of bits from set

$$\mathbb{B} \stackrel{def}{\equiv} \{0, 1\}$$

and we write down the bits from highest to lowest index. The lowest bit has index zero.

$$\forall a \in \mathbb{B}^n.\ a[n-1:0] \stackrel{def}{\equiv} a_{n-1}a_{n-2}\cdots a_0$$

Then any bit string $a \in \mathbb{B}^n$ can be interpreted as a binary number with the following value in $\mathbb{N}$.

$$\langle a[n-1:0]\rangle \stackrel{def}{\equiv} \sum_{i=0}^{n-1} a_i \cdot 2^i$$

Similarly we can interpret any bit string as an integer that is encoded in two's-complement representation. The two's-complement value of a bit string is defined as follows.

$$[a[n-1:0]] \stackrel{def}{\equiv} -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0]\rangle$$

It can be shown that in modular arithmetic $\langle a\rangle$ and $[a]$ are congruent modulo 2^n. See Section 2.2.2 in [MP00] for more information on two's complement numbers. For conversion of numbers into bit strings we use the bijections

$$bin_n : [0 : 2^n) \to \mathbb{B}^n \quad \text{and} \quad twoc_n : [-2^{n-1} : 2^{n-1}) \to \mathbb{B}^n$$

with the following properies for all $a \in \mathbb{B}^n$.

$$bin_n(\langle a\rangle) = a \qquad twoc_n([a]) = a$$

As a shorthand we allow to write X_n instead of $bin_n(X)$ for any natural number $X \in \mathbb{N}$. We define binary addition and subtraction modulo 2^n of bit strings $a, b \in \mathbb{B}^n$.

$$a +_n b \stackrel{def}{\equiv} (bin_{n+1}(\langle a\rangle + \langle b\rangle))[n-1:0] \qquad a -_n b \stackrel{def}{\equiv} (twoc_{n+1}([a] - [b]))[n-1:0]$$

Note above that the representative of a binary or two's complement number modulo 2^n can be obtained by considering only its n least significand bits. Also, since binary and two's complement values of a bit string are congruent modulo 2^n, we could have

defined addition using two's complement numbers and subtraction using binary representation of the operands. However we stick to the definitions presented above which look most natural to us.

Besides addition and subtraction we can also apply bitwise logical operations on bit strings. Let $a, b \in \mathbb{B}^n$, then we can extend any binary bitwise operator $\bullet : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ to an n-bit operator $\bullet_n : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$, such that for all $i < n$:

$$(a \bullet_n b)[i] = a_i \bullet b_i$$

In this thesis we will use $\bullet \in \{\wedge, \vee, \oplus, \overline{\vee}\}$, where $\oplus$ stands for exclusive OR (XOR) and $\overline{\vee}$ represents negated OR (NOR). We omit the subscript n where it is unambiguous.

Memory Notation

In instantiations we usually model memories m as mappings $m : \mathbb{B}^a \to \mathbb{B}^d$ of bit strings with address width a to strings with data width d. If we want to specify the content of $x \geq 1$ consecutive memory cells starting at address ad we use the following notation.

$$m_x(ad) = \begin{cases} m(ad) & : \quad x = 1 \\ m(ad +_a bin_a(x-1)) \circ m_{x-1}(ad) & : \quad \text{otherwise} \end{cases}$$

Note that the definition above as well as the remainder of this thesis uses little-endian byte order.

2 The *Cosmos* Model

In order to model multiprocessor systems later we first introduce a generic model for machines that are concurrently accessing a shared memory. We speak of a **Co**ncurrent system with **s**hared **m**emory and **o**wner**s**hip (*Cosmos*). Accesses are governed by an ownership policy guaranteeing safe memory accesses, i.e., the absence of data races on the shared memory. The ownership model builds on previous work by Cohen and Schirmer [CS09]. There it was used to ensure sequential consistency for a shared memory system with store buffers. Moreover it was designed to mirror the object-oriented ownership model of the verifier VCC on the underlying semantic layers. Here we use it to show an order reduction theorem were arbitrary interleavings of steps of different processes are reordered into schedules of blocks of steps, where all blocks start in desired interleaving-points. In the model presented below the computation units can be instantiated arbitrarily. However there is a technical well-formedness condition that we assume in the formal theory, hence any instantiations of the *Cosmos* model must be proven to fulfill it.

Below we present a formulation of the *Cosmos* model. In the subsequent chapters we will formulate and prove the reordering theorem, instantiate the *Cosmos* model with several semantics describing a computer system on different levels of abstraction, and introduce our simulation theory on top of the reduced schedules.

2.1 Signature and Instantiation Parameters

We define the *Cosmos* model by introducing a *Cosmos* machine which is a concurrent system of abstract automata operating on a common memory. We call the different automata *computation units*, or short *units*. They can be instantiated by, e.g., processors, devices, or the semantics of a higher level program. In this work, however, we assume for simplicity that all units are instantiated with the same kind of automaton. Units are only communicating via shared memory, however we have external input signals to allow for the treatment of external communication and non-determinism. We maintain information on the ownership state of individual addresses; memory is partitioned in a flexible way into sets of locally owned addresses and shared addresses. Addresses can be permanently marked as read-only. Allowing the *Cosmos* model to be instantiated for the specific application scenario, we achieve a flexible order reduction theorem which can be applied to many shared-memory concurrent systems. Below we define the signature of a *Cosmos* machine which contains all instantiable types and functions.

Definition 1 (Cosmos Machine Signature) *A* Cosmos *machine S is given by a tuple*

$$S = (\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IO}, \mathcal{IP}) \in \mathbb{S}$$

with the following components:

- $\mathcal{A}, \mathcal{V}$ *- set of memory addresses and set of memory values, any function* $m : \mathcal{A} \to \mathcal{V}$ *is called a* memory*, any partial function* $m : \mathcal{A} \rightharpoonup \mathcal{V}$ *is a* partial memory*.*
- $\mathcal{R} \subseteq \mathcal{A}$ *- set of read-only addresses (part of the ownership state)*
- nu *- the number of computation units in the machine*
- $\mathcal{U}$ *- set of computation unit states*
- $\mathcal{E}$ *- set of external inputs for the units*
- $reads : \mathcal{U} \times (\mathcal{A} \to \mathcal{V}) \times \mathcal{E} \to 2^{\mathcal{A}}$ *- the set of memory addresses read by the next step from the given unit configuration, global memory and external input. This set is called the* $reads$*-set.*
- $\delta : \mathcal{U} \times (\mathcal{A} \rightharpoonup \mathcal{V}) \times \mathcal{E} \to \mathcal{U} \times (\mathcal{A} \rightharpoonup \mathcal{V})$ *- the transition function for the units; takes unit state, a partial memory, and external input; results in a new unit state as well as another partial memory. As the input partial memory we will provide the shared memory being restricted to the* $reads$*-set of the step. The output partial memory represents the updated part of memory for the step.*
- $\mathcal{IO} : \mathcal{U} \times (\mathcal{R} \to \mathcal{V}) \times \mathcal{E} \to \mathbb{B}$ *- denotes whether the next step of the unit is an* $\mathcal{IO}$ *step.* $\mathcal{IO}$ *steps represent synchronized interactions with the environment (i.e, all other computation units), hence they include (but are not limited to) all atomic accesses to concurrently accessed data structures in memory, e.g., locks and other synchronization mechanisms. Whether the next step of a unit is an* $\mathcal{IO}$ *step, may depend on memory but only on the read-only portion.*
- $\mathcal{IP} : \mathcal{U} \times (\mathcal{R} \to \mathcal{V}) \times \mathcal{E} \to \mathbb{B}$ *- specifies the desired interleaving-points for the units, i.e., states of the computation before which we allow steps of other units to be interleaved. Whether a unit is in an interleaving-point, may depend on memory but only on the read-only portion.*

For given *Cosmos* model S, we use the shorthands $\mathcal{A}, \mathcal{V}, \mathcal{R}, \ldots$ to refer to $S.\mathcal{A}, S.\mathcal{V}, S.\mathcal{R}$, etc. As explained above every unit in the system is instantiated by the same automaton. Conceptually this is not a crucial requirement, we could also have a mapping to different computation unit models in the *Cosmos* machine, however for simplicity we do not present the more general model here. In Chap. 4 we will give several *Cosmos* machine instantiations. However to give an intuition for the intended meaning of the components consider an instantiation with a simple 32-bit multiprocessor system running in untranslated mode. For a byte-addressable memory, we set $\mathcal{A} = \mathbb{B}^{32}$ and $\mathcal{V} = \mathbb{B}^{8}$. The

read-only set $\mathcal{R}$ contains the region where the machine code of the system program resides. The unit state $\mathcal{U}$ contains all processor core registers and we use $\mathcal{E}$ to model external device interrupt signals. The reads-set always contains the address pointed to by the program counter[1] (or instruction pointer, respectively). Moreover in case of a load instruction, the targeted addresses also contribute to the reads-set. The δ-function then encodes the semantics of the underlying instruction set architecture (ISA). Depending on the programming discipline we use, the $\mathcal{IO}$ steps need to be defined: E.g., if we access shared memory only at interlocked operations (like compare-and-swap), we would choose $\mathcal{IO}$ steps to be exactly those interlocked operations. Nevertheless it is imperative that we define $\mathcal{IO}$ steps in such a way that we can later prove ownership-safety of all possible executions (see Sect. 2.6 and 3.6).

Note that in the Cohen-Schirmer theory $\mathcal{IO}$ memory instructions are denoted as *volatile* accesses. A *volatile* flag is used to denote when instructions are expected to access the shared portion of memory. However to avoid confusion with the notion of volatile accesses on the C level we rename the concept here. Actually there is a close connection between volatile and $\mathcal{IO}$ accesses, as there are certain compiler requirements for all accesses that are compiled to ISA $\mathcal{IO}$ operations. In fact all volatile accesses on the C level become $\mathcal{IO}$ accesses on the ISA level. However the two concepts are not identical. In general the notion of $\mathcal{IO}$ steps covers all operations which implement a communication with other machines. In this version of the *Cosmos* model machines can only communicate via shared memory, hence we could define an $\mathcal{IO}$ step simply to be any access to the portion of memory that is shared between all machines according to the ownership model (to be defined below). However, we decided not to do so for the following reason.

As we will see later on, the ownership policy is defined in a manner such that it is forbidden for computation units to access non-local memory outside of $\mathcal{IO}$ steps. If any access to a concurrently accessed data structure would be considered $\mathcal{IO}$, then all accesses would trivially obey the ownership policy. A different notion of memory safety for ruling out data races would be needed.

Therefore, in accordance with Cohen and Schirmer, we see the $\mathcal{IO}$ and $\mathcal{IP}$ predicates rather as annotations representing the verifier's view of the system, telling in which state a unit is expected to perform a shared memory access, in case of $\mathcal{IO}$, or where a unit's computation should be interleaved with steps of others, in case of $\mathcal{IP}$. Consequently we did not hard-code the definition of $\mathcal{IO}$ steps. However we restrict the predicates for $\mathcal{IO}$ steps and interleaving-points not to depend on writable memory. This restriction is crucial in the order reduction proof to be presented later on. It captures our intuitation that whether a unit performs an $\mathcal{IO}$ step in a given state should not directly depend on the writes of other units to shared memory. For example, in an ISA instantiation it should only depend on the machine code that is stored in read-only memory, whether a machine performs a shared memory access. Thus by the restriction of the $\mathcal{IO}$ and $\mathcal{IP}$ predicates we also rule out that $\mathcal{IO}$ steps or interleaving-points may occur due to self-modifying code.

[1] The program counter determines the next instruction to be fetched from memory.

In contrast to the Cohen-Schirmer model we confined ourselves to treat the read-only addresses as a fixed parameter of the system. In the scope of this thesis we assume that the concurrent system we are focussing on is already initialized and after that point the read-only addresses should be constant. This implies a restriction of ownership transfer after the initialization phase i.e., no read-only addresses may be acquired by machines and all released addresses must stay writable. The restriction is motivated by the reordering proof further below. If addresses may be put in or taken out of the $\mathcal{R}$ set, there needs to be a synchronization policy between the machines governing when it is safe to acquire, release and access read-only addresses. In fact, if the set of read-only addresses was not fixed, for the current ownership policy our step commutativity properties in the order reduction proof would break and we would need to resort to more complicated proof methods as seen in the Cohen-Schirmer store buffer reduction. To keep the model and the proofs simple we do not support a dynamic read-only set here. See Sect. 6.1 for a deeper discussion of the restrictions of the *Cosmos* model and its ownership policy.

2.2 Configurations

Below we define the configurations of the *Cosmos* machine which consists of the *machine state* and the *ownership state*. That means that the *Cosmos* model contains an explicit ownership model to maintain access rights of its computation units. However the ownership state does not influence the execution of the *Cosmos* machine, therefore it can be seen as specification state, or *ghost* state, respectively. Details on the ownership model are given in Sect. 2.6.

Definition 2 (Machine State) *The machine state M of a* Cosmos *model S is a pair*

$$M = (u, m) \in \mathbb{M}_S$$

where $u : \{1, \ldots, nu\} \to \mathcal{U}$ maps unit indices to their unit states and $m : \mathcal{A} \to \mathcal{V}$ is the state of the memory.

Definition 3 (Ownership State) *The ownership state $\mathcal{G}$ (ghost state) of a* Cosmos *machine S is a pair*

$$\mathcal{G} = (\mathcal{O}, \mathcal{S}) \in \mathbb{G}_S$$

where $\mathcal{O} : \{1, \ldots, nu\} \to 2^{\mathcal{A}}$ maps unit indices to the corresponding units' sets of owned addresses (owns-set) and $\mathcal{S} \subseteq \mathcal{A}$ is the set of shared writable addresses.

Now we can define the configuration of the overall *Cosmos* machine.

Definition 4 (Cosmos Machine Configuration) *A configuration C of* Cosmos *model S is given as a pair*

$$C = (M, \mathcal{G}) \in \mathbb{C}_S$$

consisting of machine state $M \in \mathbb{M}_S$ and ownership state $\mathcal{G} \in \mathbb{G}_S$.

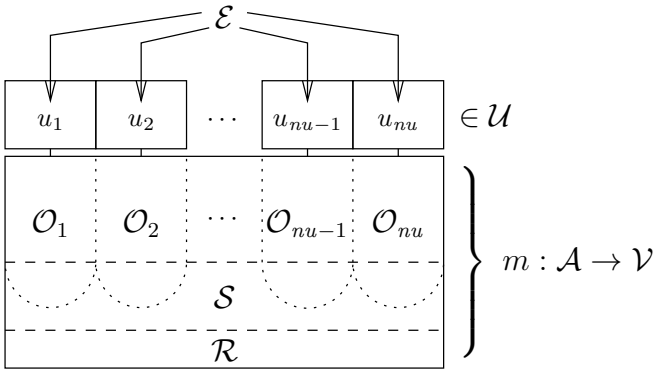

Figure 3: Illustration of the *Cosmos* machine configuration. The machine state contains the memory and the unit states which consume also external inputs. The ownership state partitions the memory address range into disjoint ownership sets, the shared-writable and the read-only addresses. Note that owned addresses may be shared or unshared (denoted by the intersection of the $\mathcal{O}_p$ and $\mathcal{S}$).

See Fig. 3 for an illustration of the overall model. For $p \in \{1, \ldots, nu\}$ we use the following shorthands:

$$\begin{aligned}
C.u_p &\equiv C.M.u(p) & C.m &\equiv C.M.m \\
C.\mathcal{O}_p &\equiv C.\mathcal{G}.\mathcal{O}(p) & C.\mathcal{S} &\equiv C.\mathcal{G}.\mathcal{S} \\
\mathit{reads}_p(C, in) &\equiv \mathit{reads}_p(C.M, in) \equiv \mathit{reads}(C.M.u(p), C.M.m, in) \\
\mathcal{IO}_p(C, in) &\equiv \mathcal{IO}_p(C.M, in) \equiv \mathcal{IO}(C.M.u(p), C.M.m|_{\mathcal{R}}, in) \\
\mathcal{IP}_p(C, in) &\equiv \mathcal{IP}_p(C.M, in) \equiv \mathcal{IP}(C.M.u(p), C.M.m|_{\mathcal{R}}, in)
\end{aligned}$$

Moreover, for defining the semantics, we need to know which addresses are written in a step of the *Cosmos* machine.

Definition 5 (Writes-set of a machine step) *For a given* Cosmos *model S with configuration $C \in \mathbb{C}_S$ and an input $in \in \mathcal{E}$ we can determine the set of written addresses in the corresponding step of machine p from the result of the delta function. This so-called $writes$-*set *of machine p is obtained with the following function.*

$$\mathit{writes}_p(C, in) = \mathrm{dom}(m') \qquad \textit{where} \qquad (u', m') = \delta(C.u_p, C.m|_{\mathit{reads}_p(C,in)}, in)$$

Note that the writes-set only depends on the part of memory that is read in the step. For the reads-set we need a similar property.

2.3 Restrictions on Instantiated Parameters

Not all parameters of a *Cosmos* model can be instantiated arbitrarily. In order to obtain a meaningful model there is in fact one constraint on the *reads*-set of *Cosmos* model computation units.

Definition 6 (Instantiation Restriction for $reads$) *By the predicate $insta_r$ we require that the $reads$-set contains all addresses upon whose memory contents it depends. For any* Cosmos *machine S let $u \in \mathcal{U}$ be a computation unit state, $m, m' \in (\mathcal{A} \to \mathcal{V})$ shared memories, and $in \in \mathcal{E}$ be a suitable input for a step of the unit. If the memory contents agree on reads-set $R = S.reads(u, m, in)$, then also the reads-set wrt. m' agrees with R.*

$$insta_r(S) \quad \equiv \quad (\, m'|_R = m|_R \implies S.reads(u, m', in) = R\,)$$

This property is needed for instantiations that incorporate a series of read accesses in one unit step. There the first reads can influence which addresses are read in later parts of the step, like in the processor instantiation example above. The reads-set must thus include all relevant addresses to determine which addresses are read. That means conversely that it only depends on the portion of memory that was read, however we can not cover this property as for the δ and $writes$ functions by providing only a partial memory representing the read memory contents. Doing so for the $reads$-function would lead to a cyclical definition.

Observe also that the property on the $reads$-set is crucial in order to be able to deduce that a machine performs the same step after reordering (by exploiting that the content of the memory region given by the $reads$-set is unchanged and thus also the same adresses are read) which results in the same update to the memory region given by the $writes$-set.

Thus, from now on, when we mention a *Cosmos* model S, we always assume that restriction $insta_r(S)$ holds.

2.4 Semantics

Units of the *Cosmos* machine execute according to their transition functions. A scheduling input decides which machine performs the next step. We assume ownership inputs that specify changes to the ownership state. These ownership inputs are given by the verification engineer annotating the program.

Definition 7 (Cosmos Model Transition Function) *For a* Cosmos *machine S we define transition function*

$$\Delta : \mathbb{C}_S \times \{1, \ldots, nu\} \times \mathcal{E} \times (2^{\mathcal{A}} \times 2^{\mathcal{A}} \times 2^{\mathcal{A}}) \to \mathbb{C}_S$$

which takes a configuration C, a scheduling input p, an external input $in \in \mathcal{E}$, the set Acq of acquired addresses, the set Loc of acquired local addresses (which should be a subset of Acq), and the set Rel of released addresses to perform a step of unit p on its state, the common memory, and the ownership state. First however we consider the transition on the machine and ownership states separately.

With $(u', m') = \delta_p(M.u(p), M.m|_{reads(M.u(p),in)}, in)$ and $m_{unchanged} = M.m|_{\mathcal{A}\setminus \mathrm{dom}(m')}$ we define transition function

$$\Delta_t(M, p, in) \equiv (M.u[p \mapsto u'], m_{unchanged} \uplus m')$$

on the machine state. Moreover with $\mathcal{O}' = (\mathcal{G}.\mathcal{O}(p) \setminus Rel) \cup Acq$ *and* $\mathcal{S}' = (\mathcal{G}.\mathcal{S} \setminus Loc) \cup Rel$ *we define the ownership transfer function:*

$$\Delta_o(\mathcal{G}, p, (Acq, Loc, Rel)) \equiv (\mathcal{G}.\mathcal{O}[p \mapsto \mathcal{O}'], \mathcal{S}')$$

Now the overall transition function for Cosmos *machine configurations is defined by:*

$$\Delta(C, p, in, (Acq, Loc, Rel)) \equiv (\Delta_t(C.M, p, in), \Delta_o(C.\mathcal{G}, p, (Acq, Loc, Rel))$$

The scheduling parameter p determines which unit is going to perform a computation step according to transition function δ consuming external input in, updating the written part of memory accordingly. The ownership transfer inputs Acq, Loc, and Rel are used to update the owned addresses of p and the set of shared-writable addresses. Acquired addresses are added to the owned addresses of p and removed from the shared addresses in case they are in the Loc set. This means that owned addresses can also be shared, which is useful in single-writer-multiple-reader scenarios [HL09]. Released addresses are moved from the owned addresses of p to the shared addresses. Note that by construction a unit cannot alter the states or ownership sets of other units.

2.5 Computations and Step Sequence Notation

In automata theory one way to reason about executions of a system is the state-based approach. There one examines sequences of machine states that where generated by applying the transition function on the initial state wrt. a given schedule and input sequence. While this formalism is useful to argue about state properties and simulation relations between states, when entering the topic of reordering this specification style becomes quite cumbersome. One problem is that one needs a permutation function encoding the reordering and relate the corresponding states to each other. As however in these states are consistent only for certain machines or components one has to introduce a multitude of equivalence relations capturing which particular part of the state is consistent for to related *Cosmos* model configurations. In earlier versions of this work it turned out that reordering proofs get quite messy in such a state-based formalism.

Alternatively one can focus on the execution steps rather than on the resulting system states. This idea is also eminent in the action-based approach used in TLA and IO-Automata [Lam94, Lyn96] and Kovalev used it to describe the reordering theorem he assumes in his doctoral thesis [Kov13]. It turns our that in this notation the formulation and proof of the reordering theorem is much more elegant than using state relations, hence we introduce a step sequence notation for our reordering and simulation theories.

The basic idea is to describe a computation not by the sequence of states it produces but by the executed sequence σ of steps from a certain alphabet. In our case the alphabet contains transition information and ownership transfer information defined as follows.

Definition 8 (Step Information) *We define the set Σ_S of step information of a* Cosmos *machine S where*

$$\alpha = (s, in, io, ip, Acq, Loc, Rel) \in \Sigma_S$$

describes a Cosmos *machine step, containing the following transition information*

- $\alpha.s \in \{1, \ldots, nu\}$ *- the scheduling parameter*
- $\alpha.in \in \mathcal{E}$ *- the external input for the step*
- $\alpha.io \in \mathbb{B}$ *- marks the step as an $\mathcal{IO}$ operation*
- $\alpha.ip \in \mathbb{B}$ *- marks the step as interleaving point of the reordered computation*

for which we introduce the type:

$$\Theta_S \equiv \{1, \ldots, nu\} \times \mathcal{E} \times \mathbb{B} \times \mathbb{B}$$

Additionally, we have the following ownership transfer information for the step:

- $\alpha.Acq \subseteq \mathcal{A}$ *- the set of acquired addresses*
- $\alpha.Loc \subseteq \mathcal{A}$ *- the set of acquired local addresses*
- $\alpha.Rel \subseteq \mathcal{A}$ *- the set of released addresses*

Ownership transfer information is of type:

$$\Omega_S \equiv 2^{\mathcal{A}} \times 2^{\mathcal{A}} \times 2^{\mathcal{A}}$$

Below we define projections, mapping step information α to transition information and ownership transfer information.

$$\alpha.t \equiv (\alpha.s, \alpha.in, \alpha.io, \alpha.ip) \qquad \alpha.o \equiv (\alpha.Acq, \alpha.Loc, \alpha.Rel)$$

Note that the step information α contains not only the necessary inputs for the *Cosmos* machine step but also the redundant flags $\alpha.ip$ and $\alpha.io$. They allow us to abstract from the intermediate configurations in a computation and to argue only about step sequences when we discuss reordering. Nevertheless, this convenience comes at the price of maintaining consistency between those flags and the values of the $\mathcal{IO}$ and $\mathcal{IP}$ predicates applied on the corresponding intermediate configurations. For $t \in \Theta_S$, $M \in \mathbb{M}_S$ and $\mathcal{X} \in \{\mathcal{IO}, \mathcal{IP}\}$ we define shorthands $\mathcal{X}(M, t) \equiv \mathcal{X}(M.u(t.s), M.m|_{\mathcal{R}}, t.in)$.

Definition 9 (Step Notation) *The notation $M \overset{t}{\mapsto} M'$ denotes that transition $t \in \Theta_S$ is executed from machine state M, resulting in M'. Additionally $t.io$ corresponds to the values of the $\mathcal{IO}$ predicate and $t.ip$ corresponds with the value of the $\mathcal{IP}$ predicate.*

$$M \overset{t}{\mapsto} M' \quad \equiv \quad M' = \Delta_t(M, t.s, t.in) \wedge \mathcal{IO}(M, t) = t.io \wedge \mathcal{IP}(M, t) = t.ip$$

For steps $\alpha \in \Sigma_S$ which include ownership transfer information we define a similar notation for the Cosmos *machine transition from configuration C into C'.*

$$C \overset{\alpha}{\mapsto} C' \;\equiv\; C.M \overset{\alpha.t}{\mapsto} C'.M \wedge C'.\mathcal{G} = \Delta_o(C.\mathcal{G}, \alpha.s, \alpha.o)$$

The definitions naturally extend to step sequences $\rho \in \Sigma_S^* \cup \Theta_S^*$ by induction:

$$X \overset{\rho}{\longmapsto} X' \;\equiv\; (\exists X'', \tau, \alpha.\ \rho = \tau\alpha \wedge X \overset{\tau}{\longmapsto} X'' \overset{\alpha}{\mapsto} X') \vee (\rho = \varepsilon \wedge X = X')$$

We use $\sigma \in \Sigma_S^*$, $\theta \in \Theta_S^*$, and $o \in \Omega_S^*$ to tell step sequences from transition sequences and ownership transfer sequences. A computation of *Cosmos* machine S can be performed with or without the ownership information since this is ghost, or specification state, respectively. A pair $(X, \rho) \in (\mathbb{C}_S \times \Sigma_S^*) \cup (\mathbb{M}_S \times \Theta_S^*)$ is then considered a *Cosmos* machine computation iff the following predicate holds:

$$comp(X, \rho) \overset{def}{\equiv} \exists X' \in \mathbb{C}_S \cup \mathbb{M}_S.\ X \overset{\rho}{\longmapsto} X'$$

We extend our step projection functions to step sequences, by mapping sequences of step information σ to transition and ownership transfer sequences.

$$\sigma.t \overset{def}{\equiv} \sigma_1.t \cdots \sigma_{|\sigma|}.t \qquad\qquad \sigma.o \overset{def}{\equiv} \sigma_1.o \cdots \sigma_{|\sigma|}.o$$

For converting a pair of transition sequence θ and ownership transfer sequence o into a step sequence σ we use the construct $\langle \theta, o \rangle$ which gives us a sequence σ such that $|\sigma| = |\theta| = |o|$ and $\sigma.t = \theta \wedge \sigma.o = o$. In particular then $\sigma = \langle \sigma.t, \sigma.o \rangle$ holds.

2.6 Ownership Policy

We use an explicit ownership model to enforce memory safety. The latter ensures that there are no memory races between the concurrently executing *Cosmos* model machines. The former allows us to distinguish between local and shared memory accesses and to use this information to justify the reordering of local machine steps later on. Below we state the restrictions we impose on the *Cosmos* model execution via the ownership policy. Later, we show that these verification conditions are sufficient to justify the desired order reduction where machine execution is interleaved only at the specified interleaving-points.

Definition 10 (Ownership Memory Access Policy) *Given a bit $io \in \mathbb{B}$, a reads-set R, a writes-set W, a set of owned addresses $\mathcal{O}$, the set of shared addresses $\mathcal{S}$, the set of read-only addresses $\mathcal{R}$, and the set of addresses owned by other machines $\overline{\mathcal{O}}$, we enforce the following ownership memory access policy given by the predicate $policy_{acc}(io, R, W, \mathcal{O}, \mathcal{S}, \mathcal{R}, \overline{\mathcal{O}})$:*

1. *local steps (i) read only owned or read-only addresses and (ii) write only owned unshared addresses*

$$/io \implies \begin{array}{ll} (i) & R \subseteq \mathcal{O} \cup \mathcal{R} \\ (ii) & W \subseteq \mathcal{O} \setminus \mathcal{S} \end{array}$$

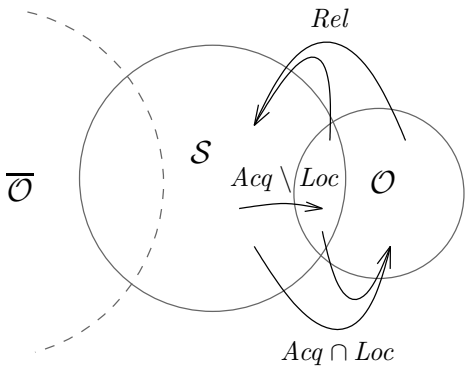

Figure 4: Illustration of the ownership state and the allowed ownership transfer for a given computation unit in the *Cosmos* model.

2. *$\mathcal{IO}$-steps may (i) read owned, shared and read-only addresses while they (ii) may write owned addresses and shared addresses which are not owned by another machine.*

$$io \implies \begin{array}{ll} (i) & R \subseteq \mathcal{O} \cup \mathcal{S} \cup \mathcal{R} \\ (ii) & W \subseteq \mathcal{O} \cup (\mathcal{S} \setminus \overline{\mathcal{O}}) \end{array}$$

Besides memory access restrictions there are also constraints on how machines can modify the ownership state. See Fig. 4 for an illustration.

Definition 11 (Ownership Transfer Policy) *Given a bit $io \in \mathbb{B}$, a set of owned addresses $\mathcal{O}$, the set of shared addresses $\mathcal{S}$, the set of addresses owned by other machines $\overline{\mathcal{O}}$, as well as the updated sets for the owned and shared addresses $\mathcal{O}'$ and $\mathcal{S}'$, we restrict ownership transfer by the predicate $policy_{trans}(io, \mathcal{O}, \mathcal{S}, \overline{\mathcal{O}}, (Acq, Loc, Rel))$.*

1. *The ownership-state may not be changed by local steps.*

$$/io \implies Acq = \emptyset \wedge Loc = \emptyset \wedge Rel = \emptyset$$

2. *For $\mathcal{IO}$-steps, the ownership-state is allowed to change as long as the step (i) acquires only addresses which are shared unowned or already owned by the executing unit and (ii) releases only owned addresses. Moreover (iii) the acquired local addresses must be a subset of the acquired addresses and (iv) one may not acquire and release the same address at a time.*

$$io \implies \begin{array}{ll} (i) & Acq \subseteq \mathcal{S} \setminus \overline{\mathcal{O}} \cup \mathcal{O} \\ (ii) & Rel \subseteq \mathcal{O} \\ (iii) & Loc \subseteq Acq \\ (iv) & Acq \cap Rel = \emptyset \end{array}$$

Observe that we restricted the original Cohen-Schirmer ownership model [CS09] in that ownership may only be modified by $\mathcal{IO}$ operations. This is not a crucial requirement, however it eases the order reduction proof effort remarkably as doing so enables splitting the proof into local commutativity arguments and the reduction of interleavings itself. See Sect. 6.1 for a detailed discussion of the subject. On the *Cosmos* model ownership state, we maintain the following ownership invariant.

Definition 12 (Ownership Invariant) *We state an ownership invariant inv on ownership state $\mathcal{G} \in \mathbb{G}_S$ of a* Cosmos *model, requiring (i) the owns-sets of different units to be mutually disjoint and (ii) that read-only addresses may not be owned or shared-writable. Moreover (iii) the complete address space is partitioned into the ownership sets as well as shared writable and read-only addresses. Moreover we set $inv(C) \equiv inv(C.\mathcal{G})$ for all $C \in \mathbb{C}_S$.*

$$\begin{aligned} inv(\mathcal{G}) \quad \equiv \quad & \textit{(i)} \quad \forall p, q.\ p \neq q \Longrightarrow \mathcal{G}.\mathcal{O}(p) \cap \mathcal{G}.\mathcal{O}(q) = \emptyset \\ & \textit{(ii)} \quad \forall p.\ \mathcal{G}.\mathcal{O}(p) \cap \mathcal{R} = \emptyset \ \wedge\ \mathcal{G}.\mathcal{S} \cap \mathcal{R} = \emptyset \\ & \textit{(iii)} \quad \mathcal{A} = \bigcup_{p \in \mathbb{N}_{np}} \mathcal{G}.\mathcal{O}(p) \cup \mathcal{G}.\mathcal{S} \cup \mathcal{R} \end{aligned}$$

We can show the following properties about safe ownership transfer.

Lemma 1 (Ownership Transfer Properies) *Given a configuration $C \in \mathbb{C}_S$ of a* Cosmos *machine S where the ownership invariant holds Let $C' = \Delta(C, p, in, (Acq, Loc, Rel))$ for given step information $(in, p, io, ip, Acq, Loc, Rel) \in \Sigma_S$. If the step obeys $policy_{trans}$ we can show (i) that addresses are only transferred between the owned addresses of p and the shared addresses, and (ii) that the new set of addresses owned by p is disjoint from the set of addresses owned by all other units $\overline{\mathcal{O}} \equiv \bigcup_{q \neq p} C.\mathcal{O}_q$.*

$$\textit{(i)} \quad C'.\mathcal{O}_p \cup C'.\mathcal{S} = C.\mathcal{O}_p \cup C.\mathcal{S} \qquad\qquad \textit{(ii)} \quad \overline{\mathcal{O}} \cap C'.\mathcal{O}_p = \emptyset$$

PROOF: By definition of Δ we have:

$$C'.\mathcal{O}_p = (C.\mathcal{O}_p \setminus Rel) \cup Acq \qquad C'.\mathcal{S} = (C.\mathcal{S} \setminus Loc) \cup Rel$$

For the first claim we use $Loc \subseteq Acq$ and $Acq \subseteq C.\mathcal{O}_p \cup C.\mathcal{S}$ from the ownership transfer safety conditions and deduce:

$$\begin{aligned} C'.\mathcal{O}_p \cup C'.\mathcal{S} &= (C.\mathcal{O}_p \setminus Rel) \cup Acq \cup (C.\mathcal{S} \setminus Loc) \cup Rel \\ &= ((C.\mathcal{O}_p \setminus Rel) \cup Rel) \cup (Acq \cup (C.\mathcal{S} \setminus Loc)) \\ &= C.\mathcal{O}_p \cup Acq \cup C.\mathcal{S} \\ &= C.\mathcal{O}_p \cup C.\mathcal{S} \end{aligned}$$

For the second claim we need to use the invariant about the disjointness of ownership sets in C, in particular we have $\overline{\mathcal{O}} \cap C.\mathcal{O}_p = \emptyset$. Then it follows:

$$\begin{aligned}
\overline{\mathcal{O}} \cap C'.\mathcal{O}_p &= \overline{\mathcal{O}} \cap ((C.\mathcal{O}_p \setminus Rel) \cup Acq) \\
&= (\overline{\mathcal{O}} \cap (C.\mathcal{O}_p \setminus Rel)) \cup (\overline{\mathcal{O}} \cap Acq) \\
&= \overline{\mathcal{O}} \cap (C.\mathcal{O}_p \setminus Rel) \\
&\subseteq \overline{\mathcal{O}} \cap C.\mathcal{O}_p \\
&= \emptyset
\end{aligned}$$

Here ownership transfer safety condition *(i)* provided that $\overline{\mathcal{O}} \cap Acq = \emptyset$. □

We subsume both the ownership access policy as well as the ownership transfer policy in a single predicate.

Definition 13 (Ownership-Safety of a Step) *We consider a step of a* Cosmos *machine S from configuration $C \in \mathbb{C}_S$ with step information $\alpha \in \Sigma_S$ to be safe with respect to the ownership model (ownership-safe) when for $R = reads_{\alpha.s}(C, \alpha.in)$, $W = writes_{\alpha.s}(C, \alpha.in)$, and $\overline{\mathcal{O}} = \bigcup_{q \neq \alpha.s} C.\mathcal{O}_q$ the following predicate is fulfilled.*

$$\begin{aligned}
safe_{step}(C, \alpha) \stackrel{def}{\equiv}\ & policy_{acc}(\alpha.io, R, W, C.\mathcal{O}_{\alpha.s}, C.\mathcal{S}, \mathcal{R}, \overline{\mathcal{O}}) \wedge \\
& policy_{trans}(\alpha.io, C.\mathcal{O}_{\alpha.s}, C.\mathcal{S}, \overline{\mathcal{O}}, \alpha.o)
\end{aligned}$$

The inductive extension of the notation for step sequences $\sigma \in \Sigma_S^*$ is straight forward.

Definition 14 (Ownership-Safety of a Computation) *For a configuration C of a* Cosmos *model S, and $\tau \in \Sigma_S^*, \alpha \in \Sigma_S$ we define*

$$\begin{aligned}
safe(C, \varepsilon) &\stackrel{def}{\equiv} inv(C) \\
safe(C, \tau\alpha) &\stackrel{def}{\equiv} safe(C, \tau) \wedge \exists C', C''.\ C \stackrel{\tau}{\longmapsto} C' \stackrel{\alpha}{\mapsto} C'' \wedge safe_{step}(C', \alpha)
\end{aligned}$$

An important property about the ownership policy stated in the following lemma is that ownership-safe steps preserve the ownership invariant.

Lemma 2 (Ownership-Safe Steps Preserve the Ownership Invariant) *For configurations $C, C' \in \mathbb{C}_S$ of a* Cosmos *model and step sequence $\sigma \in \Sigma_S^*$, we have:*

$$safe(C, \sigma) \wedge C \stackrel{\sigma}{\longmapsto} C' \implies inv(C')$$

PROOF: By induction on $n = |\sigma|$. For $n = 0$ we have $\sigma = \varepsilon$ and $C = C'$. By definition $safe(C, \varepsilon)$ collapses to $inv(C)$ hence $inv(C')$ follows directly.

In the induction step we extend σ from length $n - 1$ to n. We introduce the intermediate configuration C'' as follows.

$$C \stackrel{\sigma[1:n)}{\longmapsto} C' \stackrel{\sigma_n}{\mapsto} C''$$

Induction hypothesis yields $inv(C')$. The ownership invariants can only be broken by an unsafe modification of the ownership state in step σ_n. In particular we need to consider the set of shared addresses $C'.\mathcal{S}$ and the sets of owned addresses $C'.\mathcal{O}_p$ for some machine p. Note that by construction a machine can only modify its own ownership set, thus we have:

$$\forall q \neq \sigma_n.s.\ C'.\mathcal{O}_q = C''.\mathcal{O}_q$$

Moreover the modification of the ownership configuration is regulated by the $policy_{trans}$ predicate which is part of the definition of $safe_{step}(C', \sigma_n)$. The sets $C'.\mathcal{S}$ and $C'.\mathcal{O}_{\sigma_n.s}$ may not be changed by local steps, then invariants hold by induction hypothesis. For $\mathcal{IO}$ steps of $\sigma_n.s$ by Lemma 1 we obtain the following two necessary requirements for safe ownership transfer.

$$\textit{(i)}\quad C'.\mathcal{O}_{\sigma_n.s} \cup C'.\mathcal{S} = C''.\mathcal{O}_{\sigma_n.s} \cup C''.\mathcal{S} \qquad\qquad \textit{(ii)}\quad \overline{\mathcal{O}} \cap C''.\mathcal{O}_{\sigma_n.s} = \emptyset$$

Here $\overline{\mathcal{O}} \equiv \bigcup_{q \neq \sigma_n.s} C'.\mathcal{O}_q$ denotes the set of addresses owned by all other machines in configuration C'. As explained above $\overline{\mathcal{O}}$ is not affected by σ_n. We now prove the parts of ownership invariant $inv(C'')$ one by one.

1. $\forall p, q.\ p \neq q \implies \mathcal{O}''_p \cap \mathcal{O}''_q = \emptyset$ - If neither p nor q equals $\sigma_n.s$ the claim follows immediately from $C'.\mathcal{O}_p = C''.\mathcal{O}_p$, $C'.\mathcal{O}_q = C''.\mathcal{O}_q$, and $inv(C')$. Otherwise we assume wlog. that $p = \sigma_n.s$, thus by $C'.\mathcal{O}_q = C''.\mathcal{O}_q$ and the definition of $\overline{\mathcal{O}}$ we get $C''.\mathcal{O}_q \subseteq \overline{\mathcal{O}}$. From requirement *(ii)* we have $C''.\mathcal{O}_p \cap \overline{\mathcal{O}} = \emptyset$, thus also $C''.\mathcal{O}_p \cap C''.\mathcal{O}_q = \emptyset$.

2. $\forall p.\ C''.\mathcal{O}_p \cap \mathcal{R} = \emptyset$ - If $p \neq \sigma_n.s$ we have $C''.\mathcal{O}_p = C'.\mathcal{O}_p$ and by invariant $C'.\mathcal{O}_p \cap \mathcal{R} = \emptyset$, hence $C''.\mathcal{O}_p \cap \mathcal{R} = \emptyset$ holds. Otherwise, for $p = \sigma_n.s$, from necessary requirement *(i)* we get $C''.\mathcal{O}_{\sigma_n.s} \subseteq C'.\mathcal{O}_{\sigma_n.s} \cup C'.\mathcal{S}$, however by ownership invariant $C'.\mathcal{O}_{\sigma_n.s}$ and $C'.\mathcal{S}$ are disjoint from $\mathcal{R}$. Therefore also $C''.\mathcal{O}_{\sigma_n.s}$ is disjoint from $\mathcal{R}$.

3. $C''.\mathcal{S} \cap \mathcal{R} = \emptyset$ - This follows with the same argumentation as in the second part of the case above for $C''.\mathcal{S}$ instead of $C''.\mathcal{O}_{\sigma_n.s}$.

4. $\mathcal{A} = \bigcup_{p \in \mathbb{N}_{np}} C''.\mathcal{O}_p \cup C''.\mathcal{S} \cup \mathcal{R}$ - The invariant says that all addresses of $\mathcal{A}$ are read-only, shared-writable or owned by some machine in the system. Using $\overline{\mathcal{O}} = \bigcup_{q \neq \sigma_n.s} C'.\mathcal{O}_q = \bigcup_{q \neq \sigma_n.s} C''.\mathcal{O}_q$ this notion can be reformulated as follows:

   $$\mathcal{R} \cup C''.\mathcal{S} \cup C''.\mathcal{O}_{\sigma_n.s} \cup \overline{\mathcal{O}} = \mathcal{A}$$

   We already have $\mathcal{R} \cup C'.\mathcal{S} \cup C'.\mathcal{O}_{\sigma_n.s} \cup \overline{\mathcal{O}} = \mathcal{A}$ by $inv(C')$. By restriction *(i)* on the ownership transfer we have $C'.\mathcal{S} \cup C'.\mathcal{O}_{\sigma_n.s} = C''.\mathcal{S} \cup C''.\mathcal{O}_{\sigma_n.s}$ and the invariant on C'' stated above follows immediately. □

Thus we have proven the sanity of our safety conditions on ownership transfer. In the next chapter we will see that the ownership access and safety policy $safe_{step}$ is sufficient to prove our desired order reduction theorem.

3 Order Reduction

The ownership model introduced for *Cosmos* models allows us not only to impose safety conditions for race-free concurrent memory accesses by the units of the system. It also allows for a reordering of unit steps in the concurrent model. In the following sections we will provide theorems exploiting this fact in order to reduce the interleaving of units and justify the assumption of block scheduling when applying of sequential simulation theorems in a concurrent context.

We will first formally define the notion of interleaving-point ($\mathcal{IP}$) schedules and state the desired reduction theorem. However as we cannot prove it directly, we will introduce useful notation and lemmas, and then prove a powerful reordering theorem. Using it we will be able to show the order reduction theorem. We derive the coarse scheduling theorem presented in [CMST09] as a corollary.

3.1 Interleaving-Point Schedules

We want to consider schedules consisting of interleaved blocks of execution steps, where each block contains only steps of some unit of the *Cosmos* model. At the start of each such block the executing unit is in an interleaving-point with respect to its $\mathcal{IP}$ predicate. Such blocks we call interleaving-point blocks or $\mathcal{IP}$ blocks. Having a schedule interleaving only such $\mathcal{IP}$ blocks is convenient for Multiprocessor ISA units when we want to apply simulation theorems, e.g., use compiler consistency and go up to the C and Assembly level, later on. In this case we would choose the interleaving-points to be exactly the compiler consistency points for the unit under consideration. However the approach also applies to the modelling of systems with devices as well as preemptive threads running concurrently on the same processor. Moreover it can be used to justify the concurrent verification approach of tools like VCC.

In [CMST09] execution is reduced[1] to a coarse schedule of blocks that are interleaved at $\mathcal{IO}$ points. Usually $\mathcal{IO}$-operations like shared memory accesses start at a consistency point, however this need not be the case on all levels of abstraction[2]. Thus we prove a more general theorem and in Section 3.7 we will show that coarse scheduling with interleaving at $\mathcal{IO}$ points can be derived as a corollary of our reduction theorem.

In what follows we will show that arbitrary schedules can be reduced to $\mathcal{IP}$ schedules. Memory safety and other properties are preserved, meaning that if we prove

[1] Only the existence of a safe equivalent coarse schedule was shown but it remained open how verified safety properties transfer from coarse to fine-grained schedules.

[2] Imagine a volatile variable access in C that is compiled into an expression evaluation part and a memory access part. The former will lie between the consistency point and the $\mathcal{IO}$ operation.

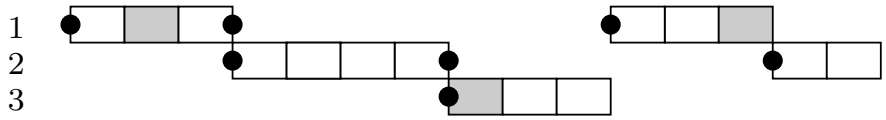

Figure 5: An example of an $\mathcal{IP}$ schedule. Steps are only interleaved at interleaving-points, represented by black circles. Empty boxes are local steps and filled boxes are $\mathcal{IO}$ steps of the corresponding coputation unit. Note that the last block of a unit in the schedule need not end in an interleaving-point.

them for all interleaving-point schedules and a given start configuration, they hold for all possible computations originating from this state. The only prerequisite is that for any computation, between two $\mathcal{IO}$-points of the same unit, this unit passes at least one interleaving-point and that in the initial state all units are in interleaving-points.

We have already introduced the predicate $\mathcal{IP}_p(C, in)$ which states that in configuration $C \in \mathbb{C}_S$ for input $in \in \mathcal{E}$ unit $p \in \mathbb{N}_{nu}$ in *Cosmos* machine S is at an interleaving-point. Now we will define the structure of our desired interleaving-point schedule. See Fig. 5 for an illustration.

Definition 15 (Interleaving-Point Schedule) *For $\rho \in \Sigma_S^* \cup \Theta_S^*$ we define the predicate*

$$\mathcal{IP}sched(\rho) \quad \equiv \quad (\,\rho = \rho'\alpha\beta \implies \mathcal{IP}sched(\rho'\alpha) \wedge ((\alpha.s = \beta.s) \vee \beta.ip)\,)$$

that expresses whether the sequence exhibits an interleaving-point schedule.

This means an $\mathcal{IP}$ schedule $\rho'\alpha$ can be extended by adding a step β of

1. the same currently running unit $\alpha.s$ or
2. another unit which is currently at an interleaving-point.

Thus in the steps of the schedule are interleaved in blocks of steps by the same unit and every block starts in an interleaving-point of its executing unit. The only exception is the first block which need not start in an interleaving-point, but this is also not necessary here and we go with the simpler definition. Note also the following property.

Lemma 3 (Trivial $\mathcal{IP}$ schedules) *Every empty or single step sequence fulfills the definition of interleaving-point schedule trivially.*

$$\forall \rho.\, |\rho| < 2 \implies \mathcal{IP}sched(\rho)$$

There are two useful properties of interleaving-point schedules to be shown.

Lemma 4 *($\mathcal{IP}$ Schedule Splitting) For any $\mathcal{IP}$ schedule ρ and any partition into two sequences τ and ω we know that both parts are $\mathcal{IP}$ schedules as well.*

$$\forall \rho, \tau, \omega.\ \mathcal{IP}sched(\rho) \wedge \rho = \tau\omega \implies \mathcal{IP}sched(\tau) \wedge \mathcal{IP}sched(\omega)$$

PROOF: By induction on $n = |\rho|$.

Induction Start: For $n = 0$ we have $\rho = \varepsilon$. The lemma trivially holds with $\tau = \varepsilon$ and $\omega = \varepsilon$ by Lemma 3 since $|\tau|, |\omega| < 2$.

Induction Hypothesis: The claim holds for any sequence $\bar{\rho}$ of length n with a partition into τ and $\bar{\omega}$, such that $\bar{\rho} = \tau\bar{\omega}$.

Induction Step: For the transition from length $n \to n + 1$ we assume a step β that is appended to $\bar{\rho}$.

$$\rho = \bar{\rho}\beta = \tau\bar{\omega}\beta = \tau\omega$$

By induction hypothesis we have $\mathcal{IP}sched(\tau)$ and $\mathcal{IP}sched(\bar{\omega})$.

1. For $\bar{\omega} = \varepsilon$ we have $\omega = \beta$ and the claim $\mathcal{IP}sched(\omega)$ holds trivially by Lemma 3 since $|\omega| < 2$.

2. If $\bar{\omega}$ is non-empty we can split it into $\bar{\omega}'$ and α as follows:

   $$\rho = \bar{\rho}\beta = \tau\bar{\omega}\beta = \tau\bar{\omega}'\alpha\beta = \tau\omega$$

   Then we know $(\alpha.s = \beta.s) \vee \beta.ip$ from $\mathcal{IP}sched(\rho)$. With this fact as well as $\mathcal{IP}sched(\bar{\omega})$, $\bar{\omega} = \bar{\omega}'\alpha$, and the definition of $\mathcal{IP}sched$ it follows $\mathcal{IP}sched(\bar{\omega}'\alpha\beta)$. We conclude $\mathcal{IP}sched(\omega)$. □

We also need a lemma on the concatenation of $\mathcal{IP}$ schedules.

Lemma 5 *($\mathcal{IP}$ Schedule Concatenation) For two $\mathcal{IP}$ schedules ρ, τ we know that their concatenation is an $\mathcal{IP}$ schedule as well if the first step of τ starts in an interleaving-point.*

$$\forall \rho, \tau.\ \mathcal{IP}sched(\rho) \wedge \mathcal{IP}sched(\tau) \wedge \tau_1.ip \implies \mathcal{IP}sched(\rho\tau)$$

PROOF: By induction on $n = |\tau|$. For $\rho = \varepsilon$ the claim is trivial because then $\rho\tau = \tau$, therefore assume $|\rho| > 0$ in the following cases.

Induction Start: For $n = 0$ we have $\rho\tau = \rho$ and the lemma trivially holds by hypothesis.

Induction Hypothesis: The claim holds for any $\mathcal{IP}$ schedule $\bar{\tau}$ of length n which is concatenated to an arbitrary interleaving-point schedule $\rho \neq \varepsilon$.

Induction Step: $n \to n + 1$ - We assume a step β such that $\tau = \bar{\tau}\beta$.

1. If $\bar{\tau} = \varepsilon$ we know from the hypothesis and $\tau_1 = \beta$ that $\beta.ip$ holds. Hence by definition of $\mathcal{IP}sched$ and hypothesis on ρ the concatenated schedule $\rho\beta$ is a interleaving-point schedule.

2. In case $\bar{\tau}$ is not empty we have $\mathcal{IP}sched(\bar{\tau})$ by definition from $\mathcal{IP}sched(\tau)$. Moreover we know $\bar{\tau}_1 = \tau_1$, hence also $\bar{\tau}_1.ip$. From induction hypothesis we thus get:

   $$\mathcal{IP}sched(\rho\bar{\tau})$$

We split $\bar{\tau}$ into $\bar{\tau}'$ and step α with subsequent properties following from premise $\mathcal{IP}sched(\tau)$.

$$\rho\tau = \rho\bar{\tau}\beta = \rho\bar{\tau}'\alpha\beta \qquad (\alpha.s = \beta.s) \vee \beta.ip$$

We see that $\rho\bar{\tau}'$ equals ρ' in the definition of $\mathcal{IP}sched$, hence $\rho\bar{\tau}'\alpha\beta$ is an $\mathcal{IP}$ schedule. We conclude $\mathcal{IP}sched(\rho\tau)$. □

We need to introduce the notions of step sub-sequences and equivalent schedule reordering in our step sequence notation.

Definition 16 (Step Subsequence Notation) *For any step or transition information sequence $\rho \in \Sigma_S^* \cup \Theta_S^*$ and unit index p we define the subsequence of steps of unit p as follows:*

$$\rho|_p \equiv \begin{cases} \alpha\tau|_p & : \quad \rho = \alpha\tau \wedge \alpha.s = p \\ \tau|_p & : \quad \rho = \alpha\tau \wedge \alpha.s \neq p \\ \varepsilon & : \quad \text{otherwise} \end{cases}$$

In the same way we introduce the $\mathcal{IO}$ step subsequence of ρ.

$$\rho|_{io} \equiv \begin{cases} \alpha\tau|_{io} & : \quad \rho = \alpha\tau \wedge \alpha.io \\ \tau|_{io} & : \quad \rho = \alpha\tau \wedge /\alpha.io \\ \varepsilon & : \quad \text{otherwise} \end{cases}$$

Lemma 6 (Subsequence Distributivity) *The subsequence operators $|_p$ and $|_{io}$ are distributive wrt. the concatenation of sequences.*

$$\forall \rho, \tau, p.\ (\rho\tau)|_p = \rho|_p\tau|_p \ \wedge\ (\rho\tau)|_{io} = \rho|_{io}\tau|_{io}$$

Both statements can be easily proven by applying the definition of $|_p$ and $|_{io}$ in an induction on the length of ρ.

In a given instantiation of a *Cosmos* model interleaving-points can be defined independently of the definition of $\mathcal{IO}$ operations. However in the reordering theorem we have the requirements that between two $\mathcal{IO}$-points a unit passes at least through one interleaving-point, and that all units start computation in an interleaving-point.

Definition 17 ($\mathcal{IOIP}$ Condition) *For any sequence $\rho \in \Sigma_S^* \cup \Theta_S^*$, predicate $\mathcal{IOIP}(\rho)$ denotes that every unit p starts in an interleaving-point and there is least one interleaving-point between any two $\mathcal{IO}$-points of p.*

$$\begin{aligned} \mathcal{IOIP}(\rho) \quad \equiv \quad & \forall \pi, p.\ \pi = \rho|_p \neq \varepsilon \implies \\ & \pi_1.ip \wedge (\forall \tau, \alpha, \varphi, \beta, \omega.\ \pi = \tau\alpha\varphi\beta\omega \wedge \alpha.io \wedge \beta.io \implies \exists i.\ \varphi_i.ip) \end{aligned}$$

Interleaving-points must be chosen by the verification engineer instantiating the model so that they fulfill this condition. To understand the necessity of its second part, it is

helpful to consider the dual of the statement which says that between two interleaving-points, there is at most one $\mathcal{IO}$ step. This reflects the well-known principle that the steps of a non-atomic operation in a concurrent program can be merged into an atomic step, as long as the operation contains at most one shared variable access [OG76].

In this work the condition is essential for the reordering, as we need to preserve the order of $\mathcal{IO}$ steps. If there were two $\mathcal{IO}$ steps of the same unit p between two subsequent interleaving-points of p, the block of steps to be constructed for p might be split in half by $\mathcal{IO}$ steps of other units which are interleaved between the two $\mathcal{IO}$ steps of p. We could not merge the two halves into one contiguous block of steps of p in this case, without changing the overall order of $\mathcal{IO}$ steps.

On the other hand, as there may only be one shared memory operation in each $\mathcal{IP}$ block, for safe execution this implies that the content of shared memory before and after that operation is the same as at the beginning, resp. the end of the block. This allows us to be more flexible concerning the position of the interleaving-points, allowing the treatment of, e.g., more efficient compilers. There we would identify the interleaving-points with compiler consistency-points. Then the $\mathcal{IOIP}$ condition would imply that $\mathcal{IO}$ steps do not have to start and end in consistency-points if the preceding and subsequent steps are local.

We will need the following two properties of the $\mathcal{IOIP}$ condition.

Lemma 7 (Properties of $\mathcal{IOIP}$ Sequences) *Let $\tau, \omega \in \Sigma_S^* \cup \Theta_S^*$ be step or transition sequences of* Cosmos *machine S, then the following properties hold.*

1. *Prefixes of schedules fulfilling the $\mathcal{IOIP}$ condition ($\mathcal{IOIP}$ sequences) are $\mathcal{IOIP}$ sequences as well.*
$$\mathcal{IOIP}(\tau\omega) \implies \mathcal{IOIP}(\tau)$$
2. *The concatenation of $\mathcal{IOIP}$ sequences fulfills the $\mathcal{IOIP}$ condition.*
$$\mathcal{IOIP}(\tau) \wedge \mathcal{IOIP}(\omega) \implies \mathcal{IOIP}(\tau\omega)$$

The detailed proofs for these statements are quite technical and they do not give much insight, hence we only provide the core arguments here.

PROOF SKETCH: The property that all units start in interleaving-points is trivially preserved when steps are removed from the end of a sequence. If two $\mathcal{IOIP}$ sequences τ and ω are concatenated then the result might contain steps of units which were not stepped in τ. However these steps also start in interleaving-points by hypothesis on ω.

The second, more important part of $\mathcal{IOIP}$ – that there is always an interleaving-point passed between two $\mathcal{IO}$ steps of the same unit – holds also on any prefix of an $\mathcal{IOIP}$ sequence because we cannot remove interleaving-points without removing also subsequent $\mathcal{IO}$ steps. Hence we cannot break the property by omitting steps at the end of a sequence. When concatenating $\mathcal{IOIP}$ sequences there may be units that have performed an $\mathcal{IO}$ step without reaching an interleaving-point until the end of τ. However, any subsequent $\mathcal{IO}$ step of these units in ω is preceded by an interleaving-point of the same unit, since by $\mathcal{IOIP}(\omega)$ all units that are scheduled in ω begin execution in an interleaving-point. □

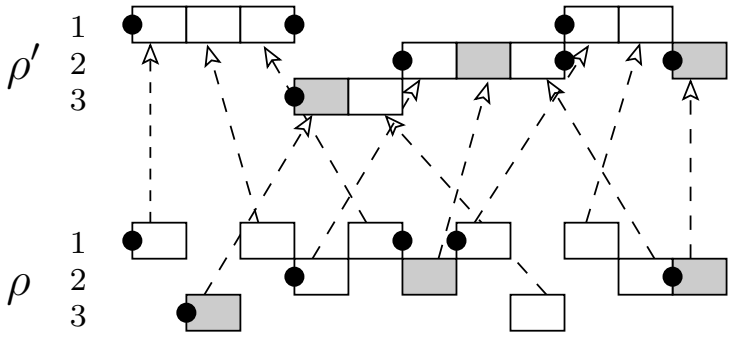

Figure 6: Example for reordering an arbitrary *Cosmos* machine schedule ρ into an $\mathcal{IP}$ schedule ρ' with $nu = 3$; empty boxes are local steps of the corresponding units, grey boxes are $\mathcal{IO}$ steps; black circles are representing interleaving-points; arrows illustrate the reordering of steps

3.2 Reordering into Interleaving-Point Schedules

One part of the order reduction theorem requires showing the existence of an equivalent interleaving-point schedule for any *Cosmos* model computation. Based on the subsequence notation, we state what it means that two step sequences are equivalently reordered.

Definition 18 (Equivalent Reordering Relation) *Given two step or transition information sequences $\rho, \rho' \in \Sigma_S^* \cup \Theta_S^*$, we consider them equivalently reordered when the $\mathcal{IO}$-step subsequence and the step subsequences of all units are the same:*

$$\rho \hat{=} \rho' \quad \equiv \quad \rho|_{io} = \rho'|_{io} \wedge \forall p \in \mathbb{N}_{nu}.\ \rho|_p = \rho'|_p$$

We also say that ρ' is an equivalent reordering of ρ and, for any starting configuration C, that (C, ρ') is an equivalently reordered computation of (C, ρ). Note that $\hat{=}$ is an equivalence relation, i.e., it is reflexive, symmetric, and transitive.

Thus the interleaving of local steps between $\mathcal{IO}$ points can be reordered arbitrarily. Only the order of $\mathcal{IO}$ points and the local order of steps for each unit must be preserved. The first condition was not present in the theorem of [Kov13], however it is an implicit property of the reordered schedules Kovalev considers, thus we make it explicit here.

An illustrating example of the reordering can be found in Figure 6. Every block in the reordered step sequence $\hat{\rho}$ starts in an interleaving-point marked by a filled circle. The dark grey boxes represent $\mathcal{IO}$ steps, e.g., shared memory accesses, which keep their order. Local steps of some unit p, represented by empty boxes, can be reordered across steps of other units. Nevertheless the order in which p executes them is preserved.

We prove a first auxiliary lemma illustrating which kind of permutation of steps is covered by the equivalent reordering relation.

Lemma 8 *Given $\alpha \in \Sigma_S \cup \Theta_S$ and a sequence $\tau \in \Sigma_S^* \cup \Theta_S^*$ such that τ contains no steps of unit $p = \alpha.s$ and either α is a local step, or τ does not contain $\mathcal{IO}$ operations, i.e., $\tau|_p = \varepsilon$ and $/\alpha.io \vee \tau|_{io} = \varepsilon$, then α can be equivalently reordered across τ.*

$$\tau\alpha \mathrel{\hat{=}} \alpha\tau$$

PROOF: By definition of relation $\hat{=}$ we need to prove that the $\mathcal{IO}$ step subsequence as well as the subsequences for all units of $\tau\alpha$ and $\alpha\tau$ must be identical. We first show the identity for the unit p using *(1)* $\tau|_p = \varepsilon$ and *(2)*Lemma 6 we deduce:

$$(\tau\alpha)|_p \overset{(2)}{=} \tau|_p\alpha|_p \overset{(1)}{=} \alpha|_p \overset{(1)}{=} \alpha|_p\tau|_p \overset{(2)}{=} (\alpha\tau)|_p$$

For any $q \neq p$ we prove the identity using *(3)* $\alpha|_q = \varepsilon$.

$$\forall q \neq p.\ (\tau\alpha)|_q \overset{(2)}{=} \tau|_q\alpha|_q \overset{(3)}{=} \tau|_q \overset{(3)}{=} \alpha|_q\tau|_q \overset{(2)}{=} (\alpha\tau)|_q$$

For the proof of the identity of the $\mathcal{IO}$ step subsequence we first assume $/\alpha.io$ thus we have *(4)* $\alpha|_{io} = \varepsilon$.

$$(\tau\alpha)|_{io} \overset{(2)}{=} \tau|_{io}\alpha|_{io} \overset{(4)}{=} \tau|_{io} \overset{(4)}{=} \alpha|_{io}\tau|_{io} \overset{(2)}{=} (\alpha\tau)|_{io}$$

In case α is an $\mathcal{IO}$ step, we have *(5)* $\tau|_{io} = \varepsilon$.

$$(\tau\alpha)|_{io} \overset{(2)}{=} \tau|_{io}\alpha|_{io} \overset{(5)}{=} \alpha|_{io} \overset{(5)}{=} \alpha|_{io}\tau|_{io} \overset{(2)}{=} (\alpha\tau)|_{io}$$

By definition we conclude $\tau\alpha \mathrel{\hat{=}} \alpha\tau$. □

There is a technical lemma saying basically that we can replay the equivalent reordering of step sequences on any ownership annotation.

Lemma 9 (Reordering Ownership Annotations) *Given two transition sequences $\theta, \theta' \in \Theta_S^*$ that are equivalently reordered, we can reorder any ownership annotation $o' \in \Omega_S^*$ for θ' into an equivalent ownership annotation o for θ. Formally:*

$$\forall \theta, \theta', o'.\ \theta \mathrel{\hat{=}} \theta' \wedge |o'| = |\theta'| \implies \exists o.\ \langle \theta, o\rangle \mathrel{\hat{=}} \langle \theta', o'\rangle$$

PROOF SKETCH: This lemma is proven by induction on $|\theta| = |\theta'|$. In the induction step the last step of o' is reordered in o to the same position as to where the last step of θ' is reordered in θ. The equivalent reordering relation $\hat{=}$ is mainly depending on the transition information of a step, in particular the scheduling component s and the io flag which determine the sub-sequences that need to be equal. The ownership anotations are irrelevant to $\hat{=}$ as long as corresponding steps have the same ownership annotations. Thus the resulting step sequences are still equivalently reordered. The detailed proof is quite technical and does not bear any further insights so we omit it here.

Now we want to show one of our main lemmas, saying that every step sequence can be equivalently reordered into an $\mathcal{IP}$ schedule. However we need another property about equivalent reorderings first.

Lemma 10 (Equivalent Reordering of $\mathcal{IOIP}$ Sequences) *The $\mathcal{IOIP}$ condition is preserved by equivalent reordering. For sequences $\rho, \rho' \in \Sigma_S^* \cup \Theta_S^*$ of* Cosmos *machine S, we have:*

$$\rho \hat{=} \rho' \wedge \mathcal{IOIP}(\rho) \implies \mathcal{IOIP}(\rho')$$

PROOF: Intuitively the claim holds because the $\mathcal{IOIP}$ condition constrains the schedules for all units separately and steps of the same unit are not permuted. Formally we have by the definition of $\mathcal{IOIP}(\rho)$:

$$\forall \pi, p.\ \pi = \rho|_p \neq \varepsilon \implies \pi_1.ip \wedge (\forall \tau, \alpha, \varphi, \beta, \omega.\ \pi = \tau\alpha\varphi\beta\omega \wedge \alpha.io \wedge \beta.io \implies \exists i.\ \varphi_i.ip)$$

Now for every p we have $\rho'|_p = \rho|_p$ by $\rho \hat{=} \rho'$, therefore also $\mathcal{IOIP}(\rho')$ holds. □

Lemma 11 (Interleaving-Point Schedule Existence) *For every step or transition sequence θ that fulfills the $\mathcal{IO}$-interleaving-point condition, we can find an interleaving-point schedule θ' which is an equivalent reordering of θ:*

$$\mathcal{IOIP}(\theta) \Longrightarrow \exists \theta'.\ \theta' \hat{=} \theta \wedge \mathcal{IP}sched(\theta')$$

PROOF: By induction on $n = |\theta|$.

Induction Start: For $n = 0$ the claim becomes trivial by $\theta' = \theta = \varepsilon$ and Lemma 3.

Induction Hypothesis: The claim holds for any sequence $\bar{\theta}$ of length n.

Induction Step: $n \to n + 1$ - Let us now consider an n+1-step sequence θ. Let $\theta = \bar{\theta}\alpha$ for some $\alpha \in \Sigma_S$. By Lemma 7.1 we have $\mathcal{IOIP}(\bar{\theta})$ and by induction hypothesis on $\bar{\theta}$ we can find an equivalently reordered $\mathcal{IP}$ schedule $\bar{\theta}'$ such that:

$$\bar{\theta}' \hat{=} \bar{\theta} \qquad \mathcal{IP}sched(\bar{\theta}')$$

With $\bar{\theta}'\alpha \hat{=} \bar{\theta}\alpha$, Lemma 10 yields $\mathcal{IOIP}(\bar{\theta}'\alpha)$. Let $p = \alpha.s$. In case $\bar{\theta}'\alpha$ is already an $\mathcal{IP}$ schedule, i.e., $\bar{\theta}'_n.s = p$ or $\alpha.ip$, we are done by setting $\theta' = \bar{\theta}'\alpha$. Otherwise, if $\bar{\theta}'_n.s \neq p$ and $/\alpha.ip$, p must have been running before, as due to the $\mathcal{IOIP}$ condition all machines start in interleaving-points. In this case we divide $\bar{\theta}'$ into subsequences π, τ, and ω as depicted in Fig. 7, i.e., we have $\bar{\theta}' = \pi\tau\omega$ with the following properties:

$$\tau \neq \varepsilon \qquad \tau|_p = \tau \qquad \tau_1.ip \qquad \forall \gamma \in tl(\tau).\ /\gamma.ip \qquad \omega \neq \varepsilon \qquad \omega|_p = \varepsilon$$

Thus τ is a block of local steps of p starting in an interleaving-point and not containing further interleaving points, while ω contains only steps from other processors, i.e., τ is the last preceding $\mathcal{IP}$ block of unit p. The $\mathcal{IOIP}$ condition and the fact that θ' is an $\mathcal{IP}$ block schedule guarantee that such a partition, in particular τ, exists, because if α does not start a interleaving-point block, it must have been running before. Observe that ω must be a non-empty sequence, otherwise $\bar{\theta}\alpha$ would be an $\mathcal{IP}$ schedule. We now perform a case distinction on the nature of the operation to be executed by p in α as depicted in Fig. 8.

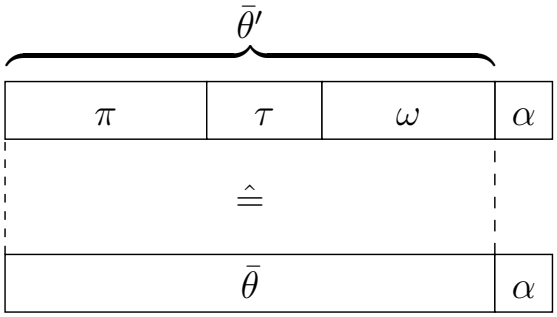

Figure 7: Application of the induction hypothesis in the induction step of Lemma 11.

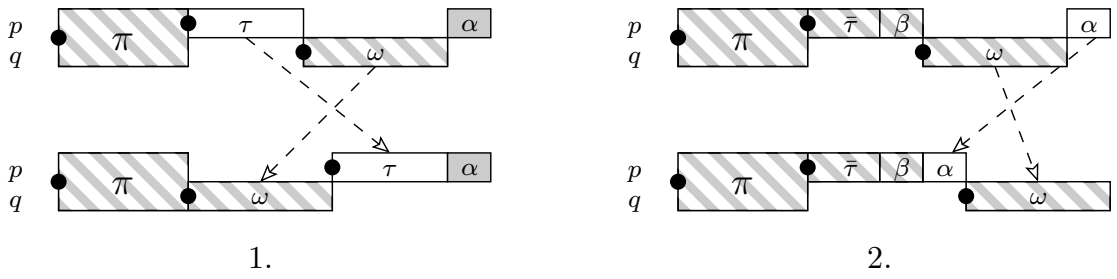

Figure 8: Illustration of reordering in the two cases of the induction step in the proof of Lemma 11. Grey boxes are $\mathcal{IO}$ steps, white boxes are local steps, and striped boxes represent a mixture of both. Index q stands for any unit but p. Block π is an $\mathcal{IP}$ schedule containing steps of all units. Note that $\tau = \bar{\tau}\beta$ may contain one $\mathcal{IO}$ step in case 2.

1. $\alpha.io$ - The n+1-st step is an $\mathcal{IO}$ operation. As we have to preserve the order of the $\mathcal{IO}$ steps we cannot reorder α across possible $\mathcal{IO}$ steps in ω, instead we move τ to the end of the sequence before step α. Note that here we have $\tau|_{io} = \varepsilon$, i.e., τ contains only local steps. This follows from the $\mathcal{IOIP}$ condition which guarantees that if p has already passed an $\mathcal{IO}$ point in $\bar{\theta}'$ then it must also have passed an interleaving-point before $\mathcal{IO}$ step α. As τ_1 is the last interleaving-point of p before α, τ cannot contain further $\mathcal{IO}$ steps. From Lemma 4 we get that π, τ and ω are interleaving-point schedules.

   $$\mathcal{IP}sched(\pi) \qquad \mathcal{IP}sched(\tau) \qquad \mathcal{IP}sched(\omega)$$

   Due to the $\mathcal{IP}sched$ property of $\bar{\theta}'$ and $\omega_1.s \neq p = \tau_{|\tau|}.s$ we have $\omega_1.ip$. Moreover we have $\tau_1.ip$. Hence we can apply Lemma 5 twice obtaining:

   $$\mathcal{IP}sched(\pi\omega\tau)$$

   Adding step α preserves the interleaving-point schedule property because $\tau_{|\tau|}.s = p = \alpha.s$. Consequently we set $\theta' = \pi\omega\tau\alpha$ and by definition it follows:

   $$\mathcal{IP}sched(\theta')$$

   As $\omega|_p = \varepsilon$ and $\tau|_p = \tau$, Lemma 6 yields:

   $$(\tau\omega)|_p = \tau|_p\omega|_p = \tau = \omega|_p\tau|_p = (\omega\tau)|_p$$

   Similarly by $\tau|_{io} = \varepsilon$ and $\tau|_q = \varepsilon$ for all $q \neq p$ we see that:

   $$(\tau\omega)|_{io} = \omega|_{io} = (\omega\tau)|_{io} \qquad (\tau\omega)|_q = \omega|_q = (\omega\tau)|_q$$

   Therefore by definition we obtain $\omega\tau \mathrel{\hat{=}} \tau\omega$ and since the remaining schedule is unchanged we conclude the last claim $\theta' \mathrel{\hat{=}} \theta$.

2. $/\alpha.io$ - In this case τ may contain an $\mathcal{IO}$ step, thus we cannot reorder τ before α. We obtain a new sequence $\theta' = \pi\tau\alpha\omega$ by reordering α to the position between τ and ω. Since $\omega|_p = \varepsilon$ and $/\alpha.io$ we can apply Lemma 8 yielding $\alpha\omega \mathrel{\hat{=}} \omega\alpha$, hence:

   $$\theta' \mathrel{\hat{=}} \theta$$

   We divide τ into a last step β and preceding steps $\bar{\tau}$, i.e., $\tau = \bar{\tau}\beta$. By Lemma 4 we have $\mathcal{IP}sched(\pi\bar{\tau}\beta)$ and $\mathcal{IP}sched(\omega)$. As $\beta.s = \alpha.s = p$ we get $\mathcal{IP}sched(\pi\bar{\tau}\beta\alpha)$ from definition. Since $\beta.s \neq \omega_1.s$ and $\bar{\theta}'$ is an $\mathcal{IP}$ block schedule we know that $\omega_1.ip$ holds. Consequently we combine $\bar{\tau}\beta\alpha$ and ω by Lemma 5 and conclude that θ' is a interleaving-point schedule, too. □

Thus we have proven that every step or transition sequence can be reordered into an $\mathcal{IP}$ schedule preserving the $\mathcal{IOIP}$ condition. However we still need to show that the reordering leads to an equivalent computation for some starting configuration. In what follows we will prove that this is the case if a *Cosmos* machine obeys the ownership policy. In particular we will show that the verification of ownership-safety and other safety properties for all interleaving-point schedules transfers down to schedules with arbitrary interleaving of unit steps. In order to achieve these results we need additional definitions and prove commutativity properties for ownership-safe steps.

3.3 Auxilliary Definitions and Lemmas

We can commute two safe steps α and β of different units if both $\alpha\beta$ and $\beta\alpha$ result in the same *Cosmos* model configuration C'' when executing the step sequences from an initial state C. Formally, the commutativity property for safe steps can be stated as follows:

$$C \overset{\alpha\beta}{\longmapsto} C'' \wedge \mathit{safe}(C, \alpha\beta) \iff C \overset{\beta\alpha}{\longmapsto} C'' \wedge \mathit{safe}(C, \beta\alpha)$$

In order to show that the statement indeed holds we need to consider intermediate configurations C_α and C_β that result from executing α, resp. β, from configuration C first. More precisely we have to show that we can execute β from C_α instead of C, or vice versa α from C_β instead of C, leading into the same state C''. This is only true if α and β do not interfere, meaning that one action does not alter data on which the other is depending. This is the same intuitive notion of interference as is used in semantics for concurrent separation logic [Bro04, Bro06].

It becomes clear that we need to relate configurations such as, e.g., C and C_α, stating for which components they are still consistent, so that we can argue that the same step β can still be executed in both states. We define the *local memory of unit* p to represent the part of memory that is read-only or unshared and owned by p. Safe local steps do not modify shared memory resources and local memory of other units. Only the state and local memory of the executing unit is affected and the pre and post states agree on all other resources. On the other hand in some circumstances all we might know is that two *Cosmos* model configurations agree about the state and local memory of some unit. Then the same local step can still be executed in both states, since local steps also do not depend on the shared unowned part of memory. We capture these relations between states by introducing two equivalence relations on *Cosmos* model configurations. They are constructed using the following basic relations denoting identity for certain *Cosmos* machine components.

Definition 19 (Cosmos Model Relations) *We define the following relations on* Cosmos *model configurations $C, D \in \mathbb{C}_S$ and a unit $p \in \mathbb{N}_{nu}$ to denote (i) the equality of p's unit state and the local memory contents, (ii) the equivalence of the ownership configurations for p, (iii) the complete equality of the ownership state, and (iv) the equality of the extent and content of the read-only and shared memory region in the system.*

$$\begin{array}{lrcl}
\textit{(i)} & C \overset{l}{\sim}_p D & \equiv & C.u_p = D.u_p \wedge C.m|_{C.\mathcal{O}_p \cup \mathcal{R}} = D.m|_{C.\mathcal{O}_p \cup \mathcal{R}} \\
\textit{(ii)} & C \overset{o}{\sim}_p D & \equiv & C.\mathcal{O}_p = D.\mathcal{O}_p \wedge C.\mathcal{O}_p \cap C.\mathcal{S} = D.\mathcal{O}_p \cap D.\mathcal{S} \\
\textit{(iii)} & C \overset{o}{\sim} D & \equiv & \forall p \in \mathbb{N}_{nu}.\ C.\mathcal{O}_p = D.\mathcal{O}_p \wedge C.\mathcal{S} = D.\mathcal{S} \\
\textit{(iv)} & C \overset{s}{\sim} D & \equiv & C.\mathcal{S} = D.\mathcal{S} \wedge C.m|_{C.\mathcal{S} \cup \mathcal{R}} = D.m|_{C.\mathcal{S} \cup \mathcal{R}}
\end{array}$$

Thus the ownership configuration of p is equivalent in systems C and D iff p owns the same addresses and these are partitioned identically into shared and local addresses. We quickly observe:

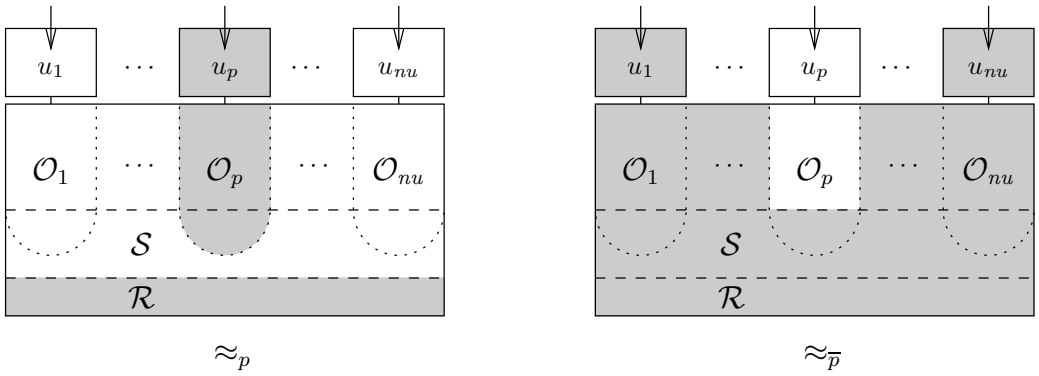

Figure 9: Illustration of p / $\overline{p}$-equivalence. Portions of the *Cosmos* machine configuration covered by the respective relation are shaded in grey.

Lemma 12 *By the definitions of $\overset{o}{\sim}$ and $\overset{o}{\sim}_p$, $C \overset{o}{\sim} D$ implies $C \overset{o}{\sim}_p D$ for all p.*

$$C \overset{o}{\sim} D \implies \forall p \in \mathbb{N}_{nu}.\, C \overset{o}{\sim}_p D$$

Using this notation we define the two equivalence relations $\approx_p$ and $\approx_{\overline{p}}$ on the set of *Cosmos* model configurations taking into account the memory contents of the ownership domain and the state components of a certain unit $p \in \mathbb{N}_{nu}$. The portions of the machine state covered by the two relations are depicted in Fig. 9.

Definition 20 (p / $\overline{p}$-Equivalence) *We define the equivalence relation*

$$C \approx_p D \;\equiv\; C \overset{l}{\sim}_p D \wedge\; C \overset{o}{\sim}_p D$$

which states that the Cosmos *model configurations C and D have the same state, local memory and ownership configuration for unit p (i.e., they are p-equivalent). Moreover we define the equivalence relation*

$$C \approx_{\overline{p}} D \quad\equiv\quad \forall q \neq p.\, C \approx_q D \wedge C \overset{s}{\sim} D \wedge C.\mathcal{O}_p = D.\mathcal{O}_p$$

which expresses identity with exception of the unit state and local memory of p (i.e., they are $\bar{p}$-equivalent).

The intention for these relations is that from p-equivalent configurations a unit p will take the same local step, and from $\bar{p}$-equivalent configurations any other unit than p will take the same local or global step, respectively. Below we prove some useful properties for the *Cosmos* model equivalence relations.

Lemma 13 *For any $p \in \mathbb{N}_{nu}$ the* Cosmos *model equivalence relations $\approx_p$ and $\approx_{\overline{p}}$ fulfill the following properties.*

1. *$C \approx_{\overline{p}} D$ implies $C \approx_q D$ for all $q \neq p$.*

2. *$C \approx_{\overline{p}} D$ implies $C \overset{o}{\sim} D$.*
3. *$C \approx_p D \wedge C \approx_{\overline{p}} D$ is equivalent to $C = D$ if $inv(C)$ holds.*

PROOF:

1. The implication follows directly from the definition of $\approx_{\overline{p}}$.
2. From definition we have $C.\mathcal{O}_p = D.\mathcal{O}_p$ and $C \overset{s}{\sim} D$ which implies $C.\mathcal{S} = D.\mathcal{S}$. From $C \approx_q D$ we get $C \overset{o}{\sim}_q D$, hence $C.\mathcal{O}_q = D.\mathcal{O}_q$ for all $q \neq p$. We conclude $C \overset{o}{\sim} D$ by definition.
3. The right-to-left implication is trivial. Vice versa from 2 we have $C.\mathcal{S} = D.\mathcal{S}$ and $C.\mathcal{O}_r = D.\mathcal{O}_r$ for all units $r \in \mathbb{N}_{nu}$. From the definitions of $\approx_p$ and $\approx_{\overline{p}}$ we get $C \approx_r D$ thus $C.u_r = D.u_r$ and consequently $C.u = D.u$. Furthermore:

   $$\forall adr, r \in \mathbb{N}_{nu}.\ adr \in C.\mathcal{O}_r \cup \mathcal{R} \cup C.\mathcal{S} \Longrightarrow C.m(adr) = D.m(adr)$$

   According to ownership invariants every address is either read-only, or shared writable, or owned. Hence also the memories $C.m = D.m$ are equal and $C = D$.

We prove two more properties of the relations $\approx_p$ and $\approx_{\overline{p}}$.

Lemma 14 *The relations $C \approx_p D$ and $C \approx_{\overline{p}} D$ are equivalence relations for any $p \in \mathbb{N}_{nu}$.*

PROOF: The relations $\overset{o}{\sim}_p$ and $\overset{s}{\sim}$ are equivalence relation because they only use unquantified equalities in their definitions. Moreover $\overset{l}{\sim}_p$ is obviously reflexive. It is only symmetric for configurations C and D in conjunction with $C.\mathcal{O}_p = D.\mathcal{O}_p$ but this is guaranteed by $C \overset{o}{\sim}_p D$. In the same way we show the transitivity, hence $C \overset{l}{\sim}_p D$ as well as $C \approx_p D$ are equivalence relations, too. $C \approx_{\overline{p}} D$ is an equivalence relation because its definition is based exclusively on equalities, $C \approx_q D$, and $C \overset{s}{\sim} D$ which are equivalence relations as shown above □

Lemma 15 *For $C, C' \in \mathbb{C}_S$ such that $C \approx_p C'$ and a transition $\alpha \in \Theta_S$ with $\alpha.s = p$ we have that $(C.M, \alpha)$ is a* Cosmos *machine computation iff $(C'.M, \alpha)$ is one.*

$$comp(C.M, \alpha) \iff comp(C'.M, \alpha)$$

PROOF: By hypothesis $C \approx_p C'$ we have $C.u_p = C'.u_p$ and $C.m|_{\mathcal{R}} = C'.m|_{\mathcal{R}}$. Since the *Cosmos* model transition function Δ can always be applied, there exists a next configuration for both C and C'. In order to show $comp(C, \alpha) = comp(C', \alpha)$ we therefore have to prove $\mathcal{IO}_{\alpha.s}(C, \alpha.in) = \mathcal{IO}_{\alpha.s}(C', \alpha.in)$ and $\mathcal{IP}_{\alpha.s}(C, \alpha.in) = \mathcal{IP}_{\alpha.s}(C', \alpha.in)$. These statements follow directly from definition and the observations above.

$$\begin{aligned}
\mathcal{IO}_{\alpha.s}(C, \alpha.in) &= \mathcal{IO}(C.u_{\alpha.s}, C.m|_{\mathcal{R}}, \alpha.in) \\
&= \mathcal{IO}(C'.u_{\alpha.s}, C'.m|_{\mathcal{R}}, \alpha.in) \\
&= \mathcal{IO}_{\alpha.s}(C', \alpha.in) \\
\mathcal{IP}_{\alpha.s}(C, \alpha.in) &= \mathcal{IP}(C.u_{\alpha.s}, C.m|_{\mathcal{R}}, \alpha.in) \\
&= \mathcal{IP}(C'.u_{\alpha.s}, C'.m|_{\mathcal{R}}, \alpha.in) \\
&= \mathcal{IP}_{\alpha.s}(C', \alpha.in)
\end{aligned}$$

□

In the remainder of this section we prove a number of lemmas on the ownership-safety of steps in related *Cosmos* configurations $C, D \in \mathbb{C}_S$ and how the relations introduced above are maintained by ownership-safe unit steps.

Lemma 16 (Reads-Set Equivalence) *When a safe local step γ of unit $p = \gamma.s$ is taken from p-equivalent configurations $C, D \in \mathbb{C}_S$ of* Cosmos *machine S, then in both computations the same set of addresses is read. The same holds for $\mathcal{IO}$ steps if additionally the shared portions of memory are identical. In both cases also the same data is read from memory in C and D.*

$$\begin{aligned} &safe_{step}(C,\gamma) \wedge C \approx_p D \wedge (\gamma.io \Rightarrow C \overset{s}{\sim} D) \implies \\ &\quad \exists R \subseteq \mathcal{A}.\ R = reads_p(C,\gamma.in) = reads_p(D,\gamma.in) \wedge C.m|_R = D.m|_R \end{aligned}$$

PROOF: We first assume that γ is a local step, i.e., $/\gamma.io$. By the safety of the step we know that p obeys $policy_{acc}$ and thus reads only from $C.\mathcal{O}_p$ and $\mathcal{R}$. These addresses have the same memory contents in both systems by the definition of $\approx_p$ and also C and D agree on the unit state of p. Let $R = reads_p(C,\gamma.in)$ then:

$$R \subseteq C.\mathcal{O}_p \cup \mathcal{R} \qquad C.m|_{C.\mathcal{O}_p \cup \mathcal{R}} = D.m|_{C.\mathcal{O}_p \cup \mathcal{R}} \qquad C.u_p = D.u_p$$

Hence it follows that also $C.m|_R = D.m|_R$ and from $insta_r(S)$ we get

$$R = reads(C.u_p, C.m, \gamma.in) = reads(D.u_p, D.m, \gamma.in)$$

which equals our claim by definition of $reads_p$. For $\mathcal{IO}$ steps $R \subseteq C.\mathcal{O}_p \cup \mathcal{R} \cup C.\mathcal{S}$ holds but by hypothesis $C \overset{s}{\sim} D$ also the shared memory region is equal in C and D.

$$C.m|_{C.\mathcal{O}_p \cup \mathcal{R} \cup C.\mathcal{S}} = D.m|_{C.\mathcal{O}_p \cup \mathcal{R} \cup C.\mathcal{S}}$$

The rest of the proof is analoguous to the first case. □

Next we show that $\bar{p}$-equivalence between configurations implies that the configurations agree on whether they fulfill the ownership invariant or they do not.

Lemma 17 (Ownership Invariant of Equivalent Configurations) *For two* Cosmos *model configurations C and $D \in \mathbb{C}_S$, we have*

$$C \overset{o}{\sim} D \implies inv(C) = inv(D)$$

PROOF: By definition of $\overset{o}{\sim}$ the ownership configurations in C and D are equivalent, i.e.:

$$C.\mathcal{S} = D.\mathcal{S} \qquad \forall p \in \mathbb{N}_{nu}.\ C.\mathcal{O}_p = D.\mathcal{O}_p$$

Moreover both C and D are configurations of the same *Cosmos* model S, hence they also agree on $\mathcal{R}$ and nu. Since the ownership invariants predicate inv only depends on the components named above our claim follows directly. □

We also need an argument about the safety transfer between equivalent system states.

Lemma 18 (Ownership-Safety in Equivalent Configurations) *Given* Cosmos *model configurations C and D as well as a step γ, the following statements hold for any $p = \gamma.s$:*

1. *If γ is a safe local step and D is p-equivalent to C, then γ is also a safe local step for D.*

$$C \approx_p D \wedge comp(C,\gamma) \wedge safe_{step}(C,\gamma) \wedge /\gamma.io \implies safe_{step}(D,\gamma) \wedge comp(D,\gamma)$$

2. *If C and D are $\bar{p}$-equivalent and γ is a step of unit $\gamma.s \neq p$ which is safe in C, then γ is also safe in D.*

$$C \approx_{\overline{p}} D \wedge comp(C,\gamma) \wedge safe_{step}(C,\gamma) \implies safe_{step}(D,\gamma) \wedge comp(D,\gamma)$$

Note that we do not need to assume that the ownership invariants are holding in C or D, however we will only apply the lemma in such cases, yielding $safe(D,\gamma)$.

PROOF: In both parts of the lemma $comp(C,\gamma)$ implies $comp(D,\gamma)$ by Lemma 15 and only $safe_{step}(D,\gamma)$ remains to be shown.

1. Machine $p = \gamma.s$ is performing a local step $C \overset{\gamma}{\mapsto} C'$ and $C \approx_p D$ holds. We know that the same memory addresses and content is read by Lemma 16. Let $R = reads_p(C,\gamma.in)$ then:

$$R = reads_p(D,\gamma.in) \qquad C.m|_R = D.m|_R \qquad C.u_p = D.u_p$$

In addition the hypothesis yields $C \overset{o}{\sim}_p D$. The definition of $safe_{step}$ contains two parts which we will treat separately to prove $safe_{step}(D,\gamma)$.

Let $\overline{\mathcal{O}}(X) = \bigcup_{q \neq p} X.\mathcal{O}_q$ for $X \in \{C, D\}$ in:

a) $policy_{acc}(\gamma.io, R, writes_p(C,\gamma.in), C.\mathcal{O}_p, C.\mathcal{S}, \mathcal{R}, \overline{\mathcal{O}}(C))$ - since the same data is read from memory and unit p has the same state in C and D, also the $writes$-sets agree.

$$\begin{aligned} \exists W \subseteq \mathcal{A}.\quad W &= writes_p(D,\gamma.in) \\ &= writes(D.u_p, D.m|_R, \gamma.in) \\ &= writes(C.u_p, C.m|_R, \gamma.in) \\ &= writes_p(C,\gamma.in) \end{aligned}$$

Furthermore, by $C \overset{o}{\sim}_p D$, p's ownership set and the shared portion of its owned addresses are identical in C and D therefore we also have

$$\begin{aligned} R \subseteq C.\mathcal{O}_p \cup \mathcal{R} &= D.\mathcal{O}_p \cup \mathcal{R} \\ W \subseteq C.\mathcal{O}_p \setminus C.\mathcal{S} &= C.\mathcal{O}_p \setminus (C.\mathcal{O}_p \cap C.\mathcal{S}) \\ &= D.\mathcal{O}_p \setminus (D.\mathcal{O}_p \cap D.\mathcal{S}) = D.\mathcal{O}_p \setminus D.\mathcal{S} \end{aligned}$$

and $policy_{acc}(\gamma.io, R, W, D.\mathcal{O}_p, D.\mathcal{S}, \mathcal{R}, \overline{\mathcal{O}}(D), \gamma.in)$ holds by definition. Note that the access policy for local steps does not depend on $\overline{\mathcal{O}}(D)$ here, hence $\overline{\mathcal{O}}(D)$ and $\overline{\mathcal{O}}(C)$ need not be related.

b) $policy_{trans}(\gamma.io, C.\mathcal{O}_p, C.\mathcal{S}, \overline{\mathcal{O}}(C), \gamma.o)$ - ownership transfer is only possible for $\mathcal{IO}$ steps, therefore we have $\gamma.o = (\emptyset, \emptyset, \emptyset)$. As we use the same step information γ with $\gamma.io = 0$ also for the step from D, we trivially get:

$$policy_{trans}(\gamma.io, D.\mathcal{O}_p, D.\mathcal{S}, \overline{\mathcal{O}}(D), \gamma.o)$$

2. Let $q = \gamma.s \neq p$ be the unit executing step γ. From $C \approx_{\overline{p}} D$ we get $C \approx_q D$ by Lemma 13.1. Moreover we have $C \overset{s}{\sim} D$, therefore Lemma 16 yields:

$$R = reads_q(C, \gamma.in) = reads_q(D, \gamma.in) \qquad C.m|_R = D.m|_R \qquad C.u_q = D.u_q$$

As above the same addresses are written when executing γ in C and D.

$$writes_q(D, \gamma.in) \;=\; writes_q(C, \gamma.in) \;=\; W$$

Since $C \overset{o}{\sim} D$ holds by Lemma 13.2, the ownership configuration is equal. Let $\overline{\mathcal{O}}(X) = \bigcup_{r \neq q} X.\mathcal{O}_r$, then in particular $\overline{\mathcal{O}}(C) = \overline{\mathcal{O}}(D)$. The ownership memory access policy only depends on the *reads*-set, the *writes*-set and the ownership state, all of which are consistent in both configurations. Also from both states the same kind of step wrt. $\gamma.io$ is executed. Since the memory access is safe in C it is also safe in D. Moreover the safety of the ownership transfer by γ from C translates directly to D. Formally we have:

$$D.\mathcal{O}_q = C.\mathcal{O}_q \qquad D.\mathcal{S} = C.\mathcal{S} \qquad \overline{\mathcal{O}}(D) = \overline{\mathcal{O}}(C)$$

Consequently, the predicates

$$policy_{acc}(\gamma.io, R, W, D.\mathcal{O}_q, D.\mathcal{S}, \mathcal{R}, \overline{\mathcal{O}}(D))$$

as well as

$$policy_{trans}(\gamma.io, D.\mathcal{O}_p, D.\mathcal{S}, \overline{\mathcal{O}}(D), \gamma.o)$$

hold by hypothesis on C. □

Furthermore we can prove two lemmas concerning unit steps on equivalent configurations wrt. $\approx_p$ and $\approx_{\overline{p}}$ which are illustrated in Figures 10 and 11.

Lemma 19 (Local Steps of a Unit) *Given a local step γ and* Cosmos *model configurations $C, C' \in \mathbb{C}_S$ with $C \overset{\gamma}{\mapsto} C'$ as well as $D, D' \in \mathbb{C}_S$ with $D \overset{\gamma}{\mapsto} D'$, we have for $p = \gamma.s$:*

$$C \approx_p D \wedge /\gamma.io \wedge safe(C, \gamma) \wedge inv(D) \implies$$

$$C' \approx_p D' \wedge C \approx_{\overline{p}} C' \wedge D \approx_{\overline{p}} D'$$

PROOF: By Lemma 18.1 the step of p in D is also safe and local, i.e., $safe(D, \gamma)$ holds. Intuitively, since the step obeys ownership memory access policy in both systems and the states are locally equivalent we see that all information accessible to the units in a local step is equal. Therefore the same local operation is executed by p in both system configurations C and D. Subsequently we prove the three claims one by one.

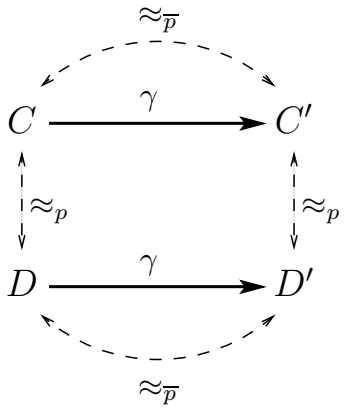

Figure 10: Illustration of Lemma 19 where $p = \gamma.s$.

1. $C' \approx_p D'$ - Expanding the definitions for $C \approx_p D$ we obtain:

$$C.u_p = D.u_p \qquad C \overset{o}{\sim}_p D \qquad C.m|_{C.\mathcal{O}_p \cup \mathcal{R}} = D.m|_{C.\mathcal{O}_p \cup \mathcal{R}}$$

By Lemma 16 we get that the same addresses and memory contents are read.

$$reads_p(D, \gamma.in) = reads_p(C, \gamma.in) = R \qquad C.m|_R = D.m|_R$$

Therefore transition function δ yields the same result and the same addresses are written.

$$\begin{aligned} \delta(C.u_p, C.m|_R, \gamma.in) &= \delta(D.u_p, D.m|_R, \gamma.in) = (u', m') \\ writes_p(D, \gamma.in) &= writes_p(C, \gamma.in) = W \end{aligned}$$

By the definition of Δ we have $C'.u_p = D'.u_p$. The ownership state is not modified by safe local steps, i.e. $C \overset{o}{\sim} C' \wedge D \overset{o}{\sim} D'$, therefore we get $C \overset{o}{\sim}_p C' \wedge D \overset{o}{\sim}_p D'$ by Lemma 12. As we have $C \overset{o}{\sim}_p D$ from $C \approx_p D$ and since $\overset{o}{\sim}_p$ is an equivalence relation we obtain:

$$C' \overset{o}{\sim}_p C \overset{o}{\sim}_p D \overset{o}{\sim}_p D' \quad \Longrightarrow \quad C' \overset{o}{\sim}_p D'$$

For the memories we have for all a in $C.\mathcal{O}_p \cup \mathcal{R}$:

$$\begin{aligned} C'.m|_{C.\mathcal{O}_p \cup \mathcal{R}}(a) &= \begin{cases} m'(a) & : \quad a \in W \\ C.m|_{C.\mathcal{O}_p \cup \mathcal{R}} & : \quad a \notin W \end{cases} \\ &= \begin{cases} m'(a) & : \quad a \in W \\ D.m|_{C.\mathcal{O}_p \cup \mathcal{R}} & : \quad a \notin W \end{cases} \\ &= D'.m|_{C.\mathcal{O}_p \cup \mathcal{R}}(a) \end{aligned}$$

Since $C'.\mathcal{O}_p \cup \mathcal{R} = C.\mathcal{O}_p \cup \mathcal{R}$ we also have $C'.m|_{C'.\mathcal{O}_p \cup \mathcal{R}} = D'.m|_{C'.\mathcal{O}_p \cup \mathcal{R}}$ and we conclude $C' \approx_p D'$.

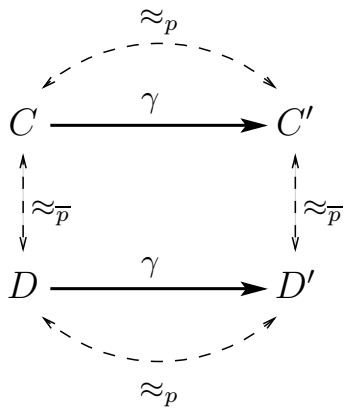

Figure 11: Illustration of Lemma 20 for any $p \neq \gamma.s$.

2. $C \approx_{\overline{p}} C'$ - Again we know that the ownership state does not change for safe local steps, i.e., $C \overset{o}{\sim} C'$ and in particular $C.\mathcal{O}_p = C'.\mathcal{O}_p$ as well as $C.\mathcal{S} = C'.\mathcal{S}$. Furthermore step γ is safe wrt. the ownership memory access policy. We prove the memory consistency condition first.

   The ownership policy forbids local writes to shared addresses, hence $C \overset{s}{\sim} C'$ holds. Moreover it is forbidden for p to write to read-only memory or addresses owned by other processors. Ownership invariants guarantee that the latter sets are disjoint from the owned addresses of p. Therefore these regions of memory are unchanged.

   $$\forall q \neq p.\ C.m|_{\mathcal{O}_q} = C'.m|_{\mathcal{O}_q} \qquad\qquad C.m|_{\mathcal{R}} = C'.m|_{\mathcal{R}}$$

   Additionally $\forall q \neq p.\ u_q = u'_q$ holds because other unit states cannot be altered by steps of p, thus $\forall q \neq p.\ C \overset{l}{\sim}_q C'$ holds by definition. By Lemma 12 and $C \overset{o}{\sim} C'$ we have $C \overset{o}{\sim}_q C'$ and thus:

   $$\forall q \neq p.\ C \approx_q C'$$

   Now our claim $C \approx_{\overline{p}} C'$ follows from the definition of $\approx_{\overline{p}}$.

3. $D \approx_{\overline{p}} D'$ - Lemma 18.1 yields $safe_{step}(D, \gamma)$ and we get $safe(D, \gamma)$ from $inv(D)$. The rest of the proof is completely analogous to the one above. □

When reordering steps of a unit p we also need to treat subsequent steps of units other than p and show that their computations are not influenced by the reordering. We have to prove among other things that the relation $\approx_{\overline{p}}$ is preserved by these so-called *environment steps*. This follows intuitively from the fact that safe computations do not depend on the local state of other units but only on data that a unit may access safely. The appropriate lemma on environment steps in its two parts is stated below.

Lemma 20 (Environment Steps) *Given a step γ and* Cosmos *model configurations $C, C' \in \mathbb{C}_S$ with $C \overset{\gamma}{\mapsto} C'$ as well as $D, D' \in \mathbb{C}_S$ with $D \overset{\gamma}{\mapsto} D'$, we have*

1. *The configuration resulting from safe step γ is p-equivalent to the intial one for all units p except for $\gamma.s$:*
$$\forall p \neq \gamma.s.\ safe(C, \gamma) \implies C \approx_p C'$$
2. *If we have ownership-safety, then $\bar{p}$-equivalence is preserved by steps of any unit but p:*
$$\forall p.\ safe(C, \gamma) \wedge C \approx_{\bar{p}} D \wedge \gamma.s \neq p \Longrightarrow C' \approx_{\bar{p}} D'$$

PROOF: We show the two parts separately.

1. For any $p \neq \gamma.s = q$ we directly have $C.u_p = C'.u_p$ and *(1)* $C.\mathcal{O}_p = C'.\mathcal{O}_p$ by construction of Δ. The step of unit q is safe, thus it also adheres to $policy_{acc}$ and cannot alter data which is owned by p. We know also that read-only addresses are never written by safe steps, therefore
$$C.m|_{\mathcal{O}_p \cup \mathcal{R}} = C'.m|_{\mathcal{O}_p \cup \mathcal{R}}$$
and $C \overset{l}{\sim}_p C'$ holds. The equivalence of p's shared and owned addresses is left to show. Using $policy_{trans}$ and $inv(C)$ we get from Lemma 1 that (2) $C.\mathcal{O}_q \cup C.\mathcal{S} = C'.\mathcal{O}_q \cup C'.\mathcal{S}$ and *(3)* $C.\mathcal{O}_p \cap C'.\mathcal{O}_q = \emptyset$. Intuitively this means that only p can share or unshare the addresses it owns. Moreover the ownership invariant (4) $C.\mathcal{O}_p \cap C.\mathcal{O}_q = \emptyset$ holds by definition of $safe(C, \gamma)$ and $inv(C)$, thus it follows:
$$\begin{aligned}
C.\mathcal{O}_p \cap C.\mathcal{S} &= \emptyset \cup (C.\mathcal{O}_p \cap C.\mathcal{S}) \\
&\overset{(4)}{=} (C.\mathcal{O}_p \cap C.\mathcal{O}_q) \cup (C.\mathcal{O}_p \cap C.\mathcal{S}) \\
&= C.\mathcal{O}_p \cap (C.\mathcal{O}_q \cup C.\mathcal{S}) \\
&\overset{(2)}{=} C.\mathcal{O}_p \cap (C'.\mathcal{O}_q \cup C'.\mathcal{S}) \\
&= (C.\mathcal{O}_p \cap C'.\mathcal{O}_q) \cup (C.\mathcal{O}_p \cap C'.\mathcal{S}) \\
&\overset{(3)}{=} \emptyset \cup (C.\mathcal{O}_p \cap C'.\mathcal{S}) \\
&\overset{(1)}{=} C'.\mathcal{O}_p \cap C'.\mathcal{S}
\end{aligned}$$
Consequently we have $C \overset{o}{\sim}_p C'$ and by definition $C \approx_p C'$ which completes our proof of the first part of the lemma.

2. From Lemma 18.2 we get the safety of the step, i.e., $safe_{step}(D, \gamma)$. By $C \overset{o}{\sim} D$ and Lemma 17 we get $inv(D)$ and thus $safe(D, \gamma)$. With Lemma 2 we obtain $inv(D')$. The definition of $\approx_{\bar{p}}$ and Lemma 13 yields:
$$\forall r \neq p.\ C \approx_r D \qquad C \overset{s}{\sim} D \qquad C \overset{o}{\sim} D$$

In particular for $V \equiv \mathcal{A} \setminus (C.\mathcal{O}_p \setminus C.\mathcal{S})$ we have $C.m|_V = D.m|_V$. Using Lemma 16 we get the equivalence of reads-sets in both states and that the same data is read by unit $q = \gamma.s \neq p$.

$$R = reads_q(C, \gamma.in) = reads_q(D, \gamma.in) \qquad C.m|_R = D.m|_R$$

Therefore by the definition of δ the same results are written, i.e., for

$$W = writes_q(C, \gamma.in) = writes_q(D, \gamma.in)$$

we have $C'.m|_W = D'.m|_W$, where $W \subseteq C.\mathcal{O}_q \cup C.\mathcal{S} \subset V$ by the safety of the step. Addresses from V which are not written preserve their memory contents by definition of Δ.

$$C'.m|_{V \setminus W} = C.m|_{V \setminus W} = D.m|_{V \setminus W} = D'.m|_{V \setminus W} \qquad C'.m|_W = D'.m|_W$$

$$\implies \quad C'.m|_{\mathcal{A} \setminus (C.\mathcal{O}_p \setminus C.\mathcal{S})} = D'.m|_{\mathcal{A} \setminus (C.\mathcal{O}_p \setminus C.\mathcal{S})}$$

Since the unit state of q and the memory contents being read are equal in C and D, transition function δ returns also the same new unit state $C'.u_q = D'.u_q$. The same holds for the ownership state, respectively.

$$C'.\mathcal{S} = (C.\mathcal{S} \cup \gamma.Rel) \setminus (\gamma.Acq \cap \gamma.Loc) = (D.\mathcal{S} \cup \gamma.Rel) \setminus (\gamma.Acq \cap \gamma.Loc) = D'.\mathcal{S}$$
$$C'.\mathcal{O}_q = (C.\mathcal{O}_q \setminus \gamma.Rel) \cup \gamma.Acq = (D.\mathcal{O}_q \setminus \gamma.Rel) \cup \gamma.Acq = D'.\mathcal{O}_q$$

Therefore we have $C' \overset{o}{\sim}_q D'$. By Lemma 1 we have $C'.\mathcal{S} \cup C'.\mathcal{O}_q = C.\mathcal{S} \cup C.\mathcal{O}_q$, hence $C'.\mathcal{S} \cup C'.\mathcal{O}_q \subseteq V$ and with $\mathcal{R} \subset V$ from $inv(C)$ we get:

$$C'.m|_{C'.\mathcal{S} \cup C'.\mathcal{O}_q \cup \mathcal{R}} = D'.m|_{C'.\mathcal{S} \cup C'.\mathcal{O}_q \cup \mathcal{R}}$$

Thus we obtain $C' \approx_q D'$ as well as $C' \overset{s}{\sim} D'$. Moreover, q cannot change the ownership set of p by construction, thus we have:

$$C'.\mathcal{O}_p = C.\mathcal{O}_p = D.\mathcal{O}_p = D'.\mathcal{O}_p$$

Only the r-equivalence between C' and D' for all $r \notin \{p, q\}$ is missing to prove our claim. By applying Lemma 20.1 on C and D for step γ and every r, the safety of step γ by q guarantees that relation $\approx_r$ is preserved for all r, i.e.:

$$C \approx_r C' \qquad D \approx_r D'$$

Thus by hypothesis $C \approx_r D$ and the transitivity of $\approx_r$ (Lemma 14) we have:

$$\forall r \notin \{p, q\}.\ C' \approx_r C \approx_r D \approx_r D' \quad \implies \quad \forall r \notin \{p, q\}.\ C' \approx_r D'$$

Hence our claim $C' \approx_{\overline{p}} D'$ follows from the definition. □

3.4 Commutativity of Local Steps

The reordering of execution schedules is based on commutativity properties. If step sequences $\alpha\beta$ and $\beta\alpha$ yield the same result then a reordering of α across β or vice versa is justified. In what follows we will present lemmas focussing on the commutativity of safe local steps, considering also property preservation. The proofs of these simple lemmas will be the only place in the reordering theory where we have to use our state relations $\approx_p$ and $\approx_{\overline{p}}$ as well as the lemmas presented above. Afterwards we can abstract from state-based reasoning and use the commutativity to argue about reordering of step sequences exclusively.

Lemma 21 (Safe Step Commutativity) *Let $C, C' \in \mathbb{C}_S$ be* Cosmos *machine configurations, and $\alpha, \beta \in \Sigma_S$ be step information with $\alpha.s \neq \beta.s$ where at least one of the steps α and β is a local step ($/\alpha.io \vee /\beta.io$), then the following statement holds:*

$$C \overset{\alpha\beta}{\longmapsto} C' \wedge \mathit{safe}(C, \alpha\beta) \quad \Longleftrightarrow \quad C \overset{\beta\alpha}{\longmapsto} C' \wedge \mathit{safe}(C, \beta\alpha)$$

PROOF: Without loss of generality we can assume $/\alpha.io$.[3] We introduce two intermediate configurations C_α and C_β and also $C_{\alpha\beta}$ and $C_{\beta\alpha}$ denoting the final configurations obtained by first executing the step encoded in α, or β respectively.

$$C_\alpha \equiv \Delta(C, \alpha.s, \alpha.in, \alpha.o) \qquad C_{\alpha\beta} \equiv \Delta(C_\alpha, \beta.s, \beta.in, \beta.o)$$
$$C_\beta \equiv \Delta(C, \beta.s, \beta.in, \beta.o) \qquad C_{\beta\alpha} \equiv \Delta(C_\beta, \alpha.s, \alpha.in, \alpha.o)$$

Note that we are not using step notation here, because it is a proof obligation to show that the reordered steps form actually a *Cosmos* machine computation, e.g., when reordering α before β we have to prove that α can be applied on C such that the io and ip flags in α are consistent with the value of the corresponding $\mathcal{IO}$ and $\mathcal{IP}$ predicates on C. Now we show the "$\Leftarrow$" and "$\Rightarrow$" implications separately. Let $\alpha.s = p$ and $\beta.s = q$.

1. "$\Leftarrow$"
$$C \overset{\beta\alpha}{\longmapsto} C' \wedge \mathit{safe}(C, \beta\alpha) \quad \Longrightarrow \quad C \overset{\alpha\beta}{\longmapsto} C' \wedge \mathit{safe}(C, \alpha\beta)$$
See Figure 12 for an illustration of the following argument. From the hypothesis we have $C \overset{\beta}{\mapsto} C_\beta \overset{\alpha}{\mapsto} C_{\beta\alpha}$, i.e., $(C, \beta\alpha)$ is a *Cosmos* machine computation and C_β as well as $C_{\beta\alpha}$ are actually reachable by β, and α respectively. The final configuration $C_{\beta\alpha}$ equals C'. Moreover we have $\mathit{safe}(C, \beta)$ and $\mathit{safe}(C_\beta, \alpha)$. Thus we can apply Lemma 20.1 and we get $C \approx_p C_\beta$ (I). Because of safety we can apply Lemma 19 on the local step α from configuration C (II). With $\gamma := \alpha$, $C := C_\beta$, $C' := C_{\beta\alpha}$, $D := C$, and $D' := C_\alpha$ in the lemma we obtain
$$C_\alpha \approx_p C_{\beta\alpha} \qquad C \approx_{\overline{p}} C_\alpha \qquad C_\beta \approx_{\overline{p}} C_{\beta\alpha}$$

[3] If α is an $\mathcal{IO}$ step, then $/\beta.io$ holds and we simply exchange the names of the steps.

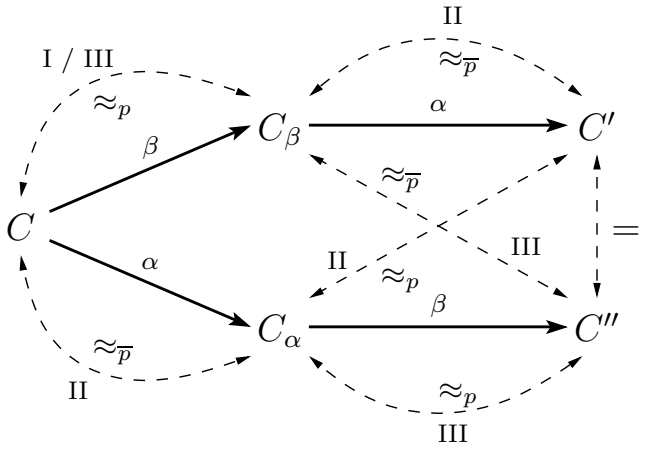

Figure 12: Proof sketch for commuting local step α of unit p forward across a step β of another unit. Step I applies Lemma 20.1 yielding $C \approx_p C_\beta$, in Step II we use Lemma 18.1 for safety transfer and 19 to execute α on C_β and C, obtaining $C' \approx_p C_\alpha$, $C_\beta \approx_{\overline{p}} C'$ and $C \approx_{\overline{p}} C_\alpha$. Lemmas 18.2 and 20.2 enable Step III where we execute β safely on C and C_α. We get $C_\alpha \approx_p C''$ as well as $C_\beta \approx_{\overline{p}} C''$ and conclude $C' \approx_p C''$ and $C' \approx_{\overline{p}} C''$ by the symmetry and transitivity of $\approx_p$ and $\approx_{\overline{p}}$. This directly implies $C' = C''$. Straight arrows indicate Δ state transitions according to α, or β resp. Dashed arrows represent the equivalence relations obtained by the proof steps as annotated.

and by Lemma 18.1 we know that the step is safe and valid, i.e., $\mathit{safe}(C, \alpha)$ and $C \overset{\alpha}{\mapsto} C_\alpha$ holds. We now have to perform β on C and C_α leading into C_β and $C_{\alpha\beta}$. With $\mathit{safe}(C, \beta)$ and $C \approx_{\overline{p}} C_\alpha$ Lemmas 20.1, 20.2, and 18.2 give us:

$$C_\alpha \approx_p C_{\alpha\beta} \qquad C_\beta \approx_{\overline{p}} C_{\alpha\beta} \qquad \mathit{safe}(C_\alpha, \beta) \qquad C_\alpha \overset{\beta}{\mapsto} C_{\alpha\beta} \qquad \text{(III)}$$

Here the variables of the lemmas were instantiated as follows: $\gamma := \beta$, $C := C$, $C' := C_\beta$, $D := C_\alpha$ and $D' := C_{\alpha\beta}$.

Using the transitivity of the relations (Lemma 14) and the definition of safe we can combine the results.

$$C_{\beta\alpha} \approx_p C_\alpha \approx_p C_{\alpha\beta} \qquad C_{\beta\alpha} \approx_{\overline{p}} C_\beta \approx_{\overline{p}} C_{\alpha\beta} \qquad \mathit{safe}(C, \alpha\beta)$$

Hence $C_{\beta\alpha} \approx_p C_{\alpha\beta}$ and $C_{\beta\alpha} \approx_{\overline{p}} C_{\alpha\beta}$ holds and by Lemma 13.3 we have:

$$C_{\beta\alpha} = C_{\alpha\beta} = C'$$

Thus we have shown that it is possible to reorder a safe and local step of one unit before an arbitrary preceding safe step of any other unit, so that ownership-safety and the effect of the steps are preserved.

2. "⇒"
$$C \overset{\alpha\beta}{\longmapsto} C' \wedge safe_P(C, \alpha\beta) \implies C \overset{\beta\alpha}{\longmapsto} C' \wedge safe_P(C, \beta\alpha)$$
The proof of the other direction is similar to the one above only the order in which lemmas are applied is changed. First Lemma 19 brings us from C with α to C_α and we obtain $C_\alpha \approx_{\overline{p}} C$. Thus we can apply β in both states and get
$$C \approx_p C_\beta \qquad C_{\alpha\beta} \approx_{\overline{p}} C_\beta \qquad safe(C, \beta) \qquad C \overset{\beta}{\mapsto} C_\beta$$
from Lemmas 20.1, 20.2, and 18.2. The first result allows us to use Lemma 19 once more executing α in C and C_β. It follows:
$$C_\alpha \approx_p C_{\beta\alpha} \qquad C_\beta \approx_{\overline{p}} C_{\beta\alpha}$$
We get the safety of the step by Lemma 18.1, therefore $safe(C, \beta\alpha)$ and $C \overset{\beta\alpha}{\longmapsto} C_{\beta\alpha}$. By application of Lemma 20.1 on the step $C_\alpha \overset{\beta}{\mapsto} C_{\alpha\beta}$ we have $C_\alpha \approx_p C_{\alpha\beta}$ and combination with above results yields:
$$C_{\alpha\beta} \approx_{\overline{p}} C_\beta \approx_{\overline{p}} C_{\beta\alpha} \qquad C_{\alpha\beta} \approx_p C_\alpha \approx_p C_{\beta\alpha}$$
With Lemma 13.3 we again conclude $C_{\alpha\beta} = C_{\beta\alpha} = C'$. □

Thus we have proven the basic commutativity arguments that safe local steps can be reordered across arbitrary safe steps by other units. Reordering across a single step can be easily extended to reordering across arbitrarily many steps.

Lemma 22 (Step Sequence Commutativity) *Let C and C' be* Cosmos *model configurations and $\sigma\alpha$ be a step sequence in such a way that no step of unit $\alpha.s$ occurs in σ, i.e. $\sigma|_{\alpha.s} = \varepsilon$ and either α is not an $\mathcal{IO}$-step or σ does not contain an $\mathcal{IO}$-step, i.e., $/\alpha.io \vee \sigma|_{io} = \varepsilon$, then:*
$$C \overset{\sigma\alpha}{\longmapsto} C' \wedge safe(C, \sigma\alpha) \iff C \overset{\alpha\sigma}{\longmapsto} C' \wedge safe(C, \alpha\sigma)$$

PROOF: by induction on $n = |\sigma|$. For $n = 0$ we have $\sigma = \varepsilon$ and there is nothing to prove, since the left and right side of the claim are identical. In the induction step from $n-1 \to n$ we let $p = \alpha.s$ and $\tau = \sigma[2:n]$, then there exists a configuration C_1 such that:
$$C \overset{\sigma_1}{\mapsto} C_1 \overset{\tau\alpha}{\longmapsto} C' \qquad safe(C, \sigma_1) \qquad safe(C_1, \tau\alpha)$$
Since $\tau|_p = \varepsilon$ and either $/\alpha.io$ or $\tau|_{io} = \varepsilon$, we can apply induction hypothesis on computation $(C_1, \tau\alpha)$ reordering it into $(C_1, \alpha\tau)$. Together with $safe(C, \sigma_1)$ we obtain:
$$C \overset{\sigma\alpha}{\longmapsto} C' \wedge safe(C, \sigma\alpha) \iff C \overset{\sigma_1\alpha\tau}{\longmapsto} C' \wedge safe(C, \sigma_1\alpha\tau)$$
Let C_2 be the configuration reached by $\sigma_1\alpha$, i.e., $C \overset{\sigma_1\alpha}{\longmapsto} C_2$. Computation $(C, \sigma_1\alpha)$ is ownership-safe and $/\alpha.io \vee /\sigma_1.io$ holds by hypothesis. Hence we can apply Lemma 21, permuting σ_1 and α.
$$C \overset{\sigma_1\alpha}{\longmapsto} C_2 \wedge safe(C, \sigma_1\alpha) \iff C \overset{\alpha\sigma_1}{\longmapsto} C_2 \wedge safe(C, \alpha\sigma_1)$$
Appending the safe computation (C_2, τ) which leads into C' to $(C, \alpha\sigma_1)$, we show that $C \overset{\sigma\alpha}{\longmapsto} C' \wedge safe(C, \sigma\alpha)$ is equivalent to $C \overset{\alpha\sigma}{\longmapsto} C' \wedge safe(C, \alpha\sigma)$. □

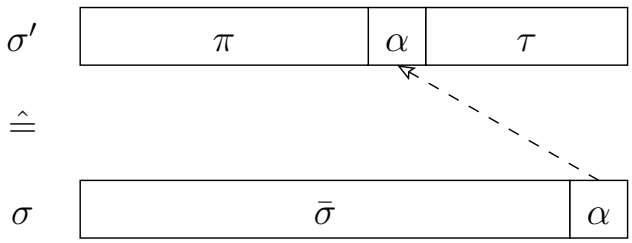

Figure 13: Illustration of reordering in the proof of Lemma 23.

3.5 Equivalent Reordering Preserves Safety

In this section we will state and prove our central reordering theorem which will allow us to reorder arbitrary schedules to interleaving-point schedules preserving the effect of the corresponding computation. For safe computations we have that all equivalently reordered computations are also safe and lead into the same configuration.

Lemma 23 (Safety of Reordered Computations) *Let $C, C' \in \mathbb{C}_S$ be* Cosmos *model configurations and let $\sigma, \sigma' \in \Sigma_S^*$ be step sequences with $C \overset{\sigma}{\longmapsto} C'$ and $\sigma \hat{=} \sigma'$ then*

$$safe(C, \sigma) \implies safe(C, \sigma') \wedge C \overset{\sigma'}{\longmapsto} C'$$

PROOF: By induction on $n = |\sigma|$.

Induction Start: For $n = 0$ the claim becomes trivial by $C = C'$ and $\sigma = \sigma' = \varepsilon$.

Induction Hypothesis: The claim holds for all sequences $\bar{\sigma}, \bar{\sigma}'$ with length n and $\bar{\sigma} = \bar{\sigma}'$, leading from configuration C into C''.

Induction Step: $n \to n + 1$ - Let $C, C' \in \mathbb{C}_S$ and σ, σ' be step sequences of length $n + 1$ with $C \overset{\sigma}{\longmapsto} C'$ and $\sigma \hat{=} \sigma'$. Let $\sigma = \bar{\sigma}\alpha$ for $\alpha \in \Sigma_S$ and C'' such that

$$C \overset{\bar{\sigma}}{\longmapsto} C'' \overset{\alpha}{\mapsto} C'$$

Let $\alpha.s = p$ and π, τ be step sequences such that $\sigma' = \pi\alpha\tau$ and *(1)* $\tau|_p = \varepsilon$ (cf. Fig. 13). This means that τ represents all the steps from σ across which α was reordered in the permutation of σ into σ'. Because equivalent reordering preserves the order of steps for each unit, we know that such a sequence τ of steps by other units than p exists. Moreover the order of $\mathcal{IO}$ steps is maintained therefore either α is local or τ does not contain any $\mathcal{IO}$ steps, i.e., *(2)* $/\alpha.io \vee \tau|_{io} = \varepsilon$.

In order to apply induction hypothesis on $\bar{\sigma}$ and $\pi\tau$ we first have to show $\bar{\sigma} \hat{=} \pi\tau$. By Lemma 8 and the symmetry of $\hat{=}$ we have $\alpha\tau \hat{=} \tau\alpha$, therefore also $\pi\alpha\tau \hat{=} \pi\tau\alpha$. From hypothesis we have $\bar{\sigma}\alpha \hat{=} \pi\alpha\tau$ and using the transitivity of $\hat{=}$ we deduce $\bar{\sigma}\alpha \hat{=} \pi\tau\alpha$. The desired equivalence $\bar{\sigma} \hat{=} \pi\tau$ follows directly.

Since (C, σ) is ownership-safe by hypothesis, also computation $(C, \bar{\sigma}) = (C, \sigma[1:n])$ is safe. Applying the induction hypothesis on $\bar{\sigma}$ and $\pi\tau$ we obtain:

$$C \overset{\pi\tau}{\longmapsto} C'' \qquad safe(C, \pi\tau)$$

By using $C'' \overset{\alpha}{\mapsto} C'$ and $safe_{step}(C'', \alpha)$ we get:

$$C \overset{\pi\tau\alpha}{\longmapsto} C' \qquad safe(C, \pi\tau\alpha)$$

Let C''' be given such that

$$C \overset{\pi}{\longmapsto} C''' \overset{\tau\alpha}{\longmapsto} C'$$

Using $C''' \overset{\tau\alpha}{\longmapsto} C'$, *(1)*, *(2)*, and $safe(C''', \tau\alpha)$ from $safe(C, \pi\tau\alpha)$, we apply Lemma 22.1, i.e., we permute α across τ while preserving the safety and effect of the computation.

$$C''' \overset{\alpha\tau}{\longmapsto} C' \qquad safe(C''', \alpha\tau)$$

With $safe(C, \pi)$ as well as $C \overset{\pi}{\longmapsto} C'''$ we have

$$C \overset{\pi\alpha\tau}{\longmapsto} C' \qquad safe(C, \pi\alpha\tau)$$

which is exactly $C \overset{\sigma'}{\longmapsto} C'$ and $safe(C, \sigma')$, hence the claim is proven. □

We have shown that ownership-safety and the effects of safe *Cosmos* machine computations are preserved by equivalent reordering. Before, we already proved that any step sequence can be equivalently reordered into an interleaving-point schedule. Thus every safe *Cosmos* machine computation is represented by an equivalent interleaving-point schedule computation and the reasoning about systems in verification can be reduced accordingly.

3.6 Reduction Theorem and Proof

In the reduction theorem the safety of all traces originating from a given starting configuration $C \in \mathbb{C}_S$ must be deduced from the safety of all interleaving-point schedules starting in the same configuration. We show not only the transfer ownership-safety but also the transfer of arbitrary verified safety properties on the concurrent system. In general, safety properties constrain finite behaviour of a *Cosmos* machine and must hold in every traversed state of a *Cosmos* machine computation. Thus we can represent them as an invariant $P : \mathbb{C}_S \to \mathbb{B}$ on the *Cosmos* machine configuration. We extend our safety predicate accordingly:

$$safe_P(C, \sigma) \overset{def}{\equiv} safe(C, \sigma) \wedge \forall C'.\, C \overset{\sigma}{\longmapsto} C' \implies P(C')$$

Then we have the following predicates denoting the verification of properties for a particular *Cosmos* model.

Definition 21 (Verified *Cosmos* machine) *We define the predicate $safety(C, P)$ which states that for all* Cosmos *machine computations starting in C we can find an ownership annotation such that the computation is safe and preserves the given property P.*

$$safety(C, P) \equiv \forall \theta.\ comp(C.M, \theta) \implies \exists o \in \Omega_S^*.\ safe_P(C, \langle \theta, o \rangle)$$

We also define the predicate $safety_{\mathcal{IP}}(C, P)$ which expresses the same notion of verification for all $\mathcal{IP}$ schedule computations:

$$safety_{\mathcal{IP}}(C, P) \equiv \forall \theta.\ \mathcal{IP}sched(\theta) \wedge comp(C.M, \theta) \implies \exists o \in \Omega_S^*.\ safe_P(C, \langle \theta, o \rangle)$$

Additionally all $\mathcal{IP}$ schedules starting in C need to fulfill the $\mathcal{IOIP}$ condition.

$$\mathcal{IOIP}_{\mathcal{IP}}(C) \equiv \forall \theta.\ \mathcal{IP}sched(\theta) \wedge comp(C.M, \theta) \implies \mathcal{IOIP}(\theta)$$

Thus we consider a *Cosmos* model ownership-safe if any transition sequence can be annotated with an ownership transfer sequence such that the ownership policy is obeyed. In order to see that this verification methodology in deed excludes memory races we distinguish shared and local addresses.

Shared addresses may only be accessed by $\mathcal{IO}$ operations which represent atomic shared variable accesses that are implemented, e.g., by synchronization primitives on the processor hardware level, or by assignments to volatile variables on the C level. Hence the ownership policy requires all units to access shared memory locations in a synchronized fashion.

For local steps stronger access rules are established and only owned memory maybe modified. However there could be races on local memory due to ownership transfer. Nevertheless ownership can not transfer directly between units and ownership transfer is bound to $\mathcal{IO}$ steps. Hence ownership of some address first has to be released by the former owner to shared memory before it can be acquired by the new owner, requiring two $\mathcal{IO}$ operations in total. Therefore the ownership policy enforces synchronization also for ownership transfer.

For the sake of an alternative explanation we define a computation to be racy, if there exists an equivalently reordered computation resulting in a different end configuration. For ownership-safe computations we know however by Lemma 23 that any equivalently reordered computation leads into the same end configuration. Thus the ownership-safety of all computations excludes memory races.

Using the definitions from above the interleaving-point schedule reduction theorem can then be stated as follows.

Theorem 1 ($\mathcal{IP}$ Schedule Order Reduction) *For a configuration C of a* Cosmos *machine S where all $\mathcal{IP}$ schedule computations originating in C fulfill the $\mathcal{IOIP}$ condition, we can deduce safety property P and ownership-safety on all possible computations from the verification of these properties on all $\mathcal{IP}$ schedules.*

$$safety_{\mathcal{IP}}(C, P) \wedge \mathcal{IOIP}_{\mathcal{IP}}(C) \implies safety(C, P)$$

Note that we only require safety on the order-reduced *Cosmos* model model. In contrast to the existing order reduction theorems of the literature, the commutativity of local steps is not assumed to be justified on the unreduced level – instead, we exploit that we can propagate the ownership-safety of $\mathcal{IP}$ schedules to all schedules of the unreduced model in order to prove this overall theorem. To this end we need to show that every *Cosmos* machine computation can be represented by an equivalently reordered $\mathcal{IP}$ schedule computation.

Lemma 24 (Coverage) *From $safety_{\mathcal{IP}}(C, P)$ and $\mathcal{IOIP}_{\mathcal{IP}}(C)$ it follows that for any* Cosmos *machine computations $(C.M, \theta)$ the $\mathcal{IOIP}$ condition is fulfilled and any equivalently reordered $\mathcal{IP}$ schedule can be executed from C.*

$$\begin{aligned} &safety_{\mathcal{IP}}(C, P) \wedge \mathcal{IOIP}_{\mathcal{IP}}(C) \wedge comp(C.M, \theta) \implies \\ &\qquad \mathcal{IOIP}(\theta) \wedge (\forall \theta'.\ \theta \mathrel{\hat{=}} \theta' \wedge \mathcal{IP}sched(\theta') \implies comp(C.M, \theta')) \end{aligned}$$

PROOF: by induction on length $n = |\theta|$ of the computation. For empty schedules the claim holds trivially.

In the induction step from $n-1 \to n$ we assume that $\theta = \bar{\theta}\alpha$ and by induction hypothesis we have $\mathcal{IOIP}(\bar{\theta})$. Using Lemma 11 we get an equivalently reordered $\mathcal{IP}$ schedule $\hat{\theta}$ for which the $\mathcal{IOIP}$ condition holds by hypothesis.

$$\hat{\theta} \mathrel{\hat{=}} \bar{\theta} \qquad \mathcal{IP}sched(\hat{\theta}) \qquad \mathcal{IOIP}(\hat{\theta})$$

By hypothesis there also exists an ownership annotation $\hat{o} \in \Omega_S^*$ such that $safe(C, \langle \hat{\theta}, \hat{o} \rangle)$ and by Lemma 9 we obtain a reordered ownership transfer sequence $\bar{o}$ for $\bar{\theta}$. With Lemma 23 we have that $\langle \bar{\theta}, \bar{o} \rangle$ is safe and leads into the same configuration as $\langle \hat{\theta}, \hat{o} \rangle$.

$$safe(C, \langle \bar{\theta}, \bar{o} \rangle) \qquad C \overset{\langle \hat{\theta}, \hat{o} \rangle}{\longmapsto} C' \implies C \overset{\langle \bar{\theta}, \bar{o} \rangle}{\longmapsto} C'$$

Then $comp(C.M, \bar{\theta}\alpha)$ implies $comp(C.M, \hat{\theta}\alpha)$ because the same intermediate machine state $C'.M$ is passed before executing α. If $\hat{\theta}\alpha$ is an $\mathcal{IP}$ schedule, then we are done because then we have $\mathcal{IOIP}(\hat{\theta}\alpha)$ by hypothesis and $\mathcal{IOIP}(\theta)$ by $\theta \mathrel{\hat{=}} \hat{\theta}\alpha$ and Lemma 10. Otherwise we need to perform a case split. In what follows let $p = \alpha.s$.

1. $\hat{\theta}|_p = \varepsilon \wedge /\alpha.ip$ — step α is the first step of unit p but it does not start in an interleaving-point. However it is excluded that this case can occur because of the following argument. Since $\hat{\theta}\alpha$ is a computation we have also $/\mathcal{IP}_p(C', \alpha.in)$ and by inductive application of Lemma 20.1 on the complete previous computation $(C, \langle \hat{\theta}, \hat{o} \rangle)$, that is safe, we get $C \approx_p C'$. By Lemma 15 we have $comp(C.M, \alpha)$, i.e., α could be executed also in the beginning of the computation. Moreover $(C.M, \alpha)$ is an $\mathcal{IP}$ schedule computation by Lemma 3, hence hypothesis $\mathcal{IOIP}_{\mathcal{IP}}(C)$ gives us $\mathcal{IOIP}(\alpha)$ which implies $\alpha.ip$, contradicting our assumption $/\alpha.ip$.

2. $\exists \psi, \beta, \omega.\ \langle \hat{\theta}, \hat{o} \rangle = \psi\beta\omega \wedge \beta.s = p \wedge \omega|_p = \varepsilon$ — we search the preceding step β of unit p in $\hat{\theta}$ (cf. Fig. 14, lower part). Let C'' be the resulting intermediate configuration.

$$C \overset{\psi\beta}{\longmapsto} C'' \overset{\omega}{\longmapsto} C'$$

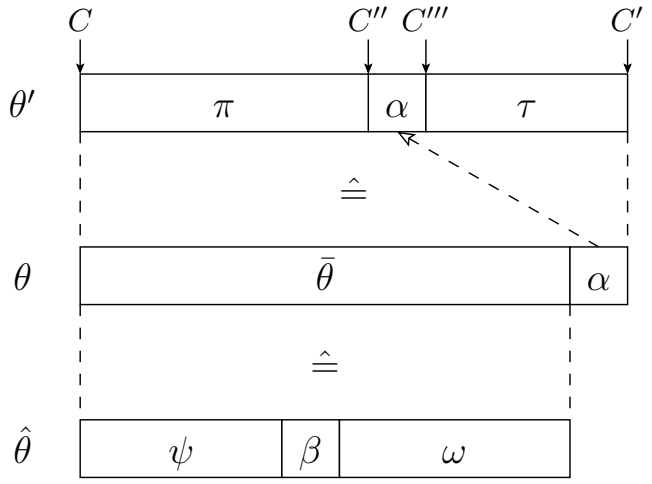

Figure 14: Reorderings in the induction step of the proof of Lemma 24.

By inductive application of Lemma 20.1 on (C'', ω), we obtain $C'' \approx_p C'$, thus also $comp(C'', \alpha)$ and $comp(C, \psi\beta\alpha)$. By the definition of $\mathcal{IP}$ schedules we have $\mathcal{IP}sched(\psi\beta\alpha)$, therefore by hypothesis also $\mathcal{IOIP}(\psi\beta\alpha)$ holds. From $\theta \hat{=} \hat{\theta}\alpha$ we have $\theta|_q = \hat{\theta}|_q$ for all units $q \neq p$ and by $\omega|_p = \varepsilon$ also $\theta|_p = (\psi\beta\alpha).t|_p$. Then from $\mathcal{IOIP}(\hat{\theta})$ and $\mathcal{IOIP}(\psi\beta\alpha)$ we conclude $\mathcal{IOIP}(\theta)$ because the $\mathcal{IOIP}$ condition constrains the subsequences of each unit in θ separately.

To prove the second claim we again distinguish whether p has been running before or not. In the latter case we are finished because then we have $\alpha.ip$ as explained above. From definition it follows that $\hat{\theta}\alpha$ is an $\mathcal{IP}$ schedule and we also have $\theta \hat{=} \hat{\theta}\alpha$ as well as $comp(C.M, \hat{\theta}\alpha)$, which proves our claim.

If p was running before we assume an $\mathcal{IP}$ schedule $\theta' = \pi\alpha\tau$ with $\theta \hat{=} \theta'$ and the properties $\pi_{|\pi|}.s = p$, $\tau|_p = \varepsilon$, as well as $/\alpha.io \vee \tau|_{io} = \varepsilon$ (cf. Fig. 14, upper part). It remains to be proven that $(C.M, \pi\alpha\tau)$ is a *Cosmos* machine computation.

We can show $\bar{\theta} \hat{=} \pi\tau$ as in the proof of Lemma 23. Moreover, by Lemma 4, $\pi\alpha$ and τ are $\mathcal{IP}$ schedules and we have $\tau_1.ip$ in case τ is non-empty, because τ starts with a step by a different unit than p. Omitting α from $\pi\alpha$ yields $\mathcal{IP}sched(\pi)$ by another invocation of Lemma 4. Then using Lemma 5 we get $\mathcal{IP}sched(\pi\tau)$.

Applying the induction hypothesis on $\bar{\theta} \hat{=} \pi\tau$ we obtain that $(C.M, \pi\tau)$ is a *Cosmos* machine computation. With Lemma 9 we reorder $\bar{o}$ into an ownership annotation $o_\pi o_\tau$, such that $o_\pi \in \Omega_S^{|\pi|}$, $o_\tau \in \Omega_S^{|\tau|}$, and $\langle \bar{\theta}, \bar{o} \rangle \hat{=} \langle \pi\tau, o_\pi o_\tau \rangle$. By Lemma 23 the computation is safe and leads into configuration C'.

$$safe(C, \langle \pi\tau, o_\pi o_\tau \rangle) \qquad C \overset{\langle \pi\tau, o_\pi o_\tau \rangle}{\longmapsto} C'$$

We redefine C'' to be the configuration reached by $\langle \pi, o_\pi \rangle$ from C, i.e., $C \overset{\langle \pi, o_\pi \rangle}{\longmapsto} C''$. Then by inductive application of Lemma 20.1 on the safe computation $(C'', \langle \tau, o_\tau \rangle)$ we get

$C'' \approx_p C'$. Lemma 15 yields $comp(\pi\alpha)$ which is an $\mathcal{IP}$ schedule computation because the last step of π and α are executed by the same unit p. Then by hypothesis there exists a safe annotation o_α for α, i.e., $safe(C, \langle \pi\alpha, o_\pi o_\alpha \rangle)$ holds. Let C''' be the configuration reached by executing $\langle \alpha, o_\alpha \rangle$ in C''.

In case α is a local step, by Lemma 19 we directly get $C'' \approx_{\overline{p}} C'''$. By construction we have $comp(C'', \langle \tau, o_\tau \rangle)$ and inductive use of Lemma 15 and Lemma 20.2 justifies the execution of $\langle \tau, o_\tau \rangle$ from C'''. Lemma 20.2 is needed here to maintain $\approx_{\overline{p}}$ between the corresponding intermediate configurations. Thus we get $comp(C'''.M, \tau)$ in this case.

If we have $\alpha.io$, then all steps in τ are local by construction. We get $C'' \approx_q C'''$ for all $q \neq p$ by application of Lemma 20.1. Again we use Lemma 15 inductively to justify the execution of $\langle \tau, o_\tau \rangle$ from C''' but this time $\approx_q$ between the corresponding intermediate computations is maintained by Lemma 19 for all q.

Again we get $comp(C'''.M, \tau)$ and with $comp(C.M, \pi\alpha)$ we deduce $comp(C.M, \theta')$ for $\theta' = \pi\alpha\tau$, which was our last claim. □

Now we can prove the order reduction theorem.

PROOF OF THEOREM 1: Given a computation $(C.M, \theta)$, we need to show that there exists an ownership annotation o such that $(C, \langle \theta, o \rangle)$ is safe. By Lemma 24 we obtain an equivalent interleaving-point schedule θ', such that $\theta \hat{=} \theta'$, $\mathcal{IP}sched(\theta')$, and $comp(C.M, \theta')$ holds. Then by $safety_{\mathcal{IP}}$ there exists an o' such that $\langle \theta', o' \rangle$ is safe. According to Lemma 9 we can reorder o' into some ownership transfer sequence o such that we have $\langle \theta, o \rangle \hat{=} \langle \theta', o' \rangle$, using the same permutation of steps as in the reordering of θ' into θ. By the symmetry of $\hat{=}$ we also have $\langle \theta', o' \rangle \hat{=} \langle \theta, o \rangle$ and using Lemma 23 as well as $safe_P(C, \langle \theta', o' \rangle)$ we conclude the safety of $(C, \langle \theta, o \rangle)$ which results in the same configuration C' where $P(C')$ holds. □

This finishes our order reduction theory. We have shown that from now on we can treat *Cosmos* machine computations at a granularity where unit execution is only interleaved at interleaving-points that can be specified arbitrarily by the verification engineer. The only additional verification conditions are, that between two $\mathcal{IO}$ points a unit always passes at least one interleaving-point, each unit starts in an interleaving point, and that for all computations with interleaving-point schedules we can find an ownership annotation such that the ownership policies for memory access and ownership transfer are obeyed. We will show later how these verification conditions can be discharged in a framework of simulation theorems between different layers of abstraction, such that ideally properties are only proven about $\mathcal{IP}$ schedules of the highest level of abstraction (cf. Section 5.4).

In order to give a first example of the generality and usability of our approach we show that Theorem 1 allows for the justification of a specific kind of order reduction that is often applied in the analysis of shared memory concurrent systems.

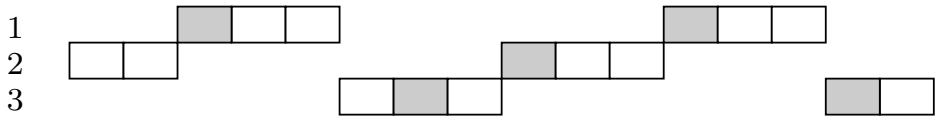

Figure 15: An example of an $\mathcal{IO}$-block schedule θ for three computation units. Empty boxes are local steps, while filled boxes represent $\mathcal{IO}$ steps.

3.7 Coarse $\mathcal{IO}$-Block Scheduling

The order reduction theorem presented in [CMST09] proposes a very coarse way of scheduling. In the concurrent computation of a system unit steps are executed in blocks that start with $\mathcal{IO}$ steps or with the first step of a unit. Thus, after some initial phase, blocks of steps of different computation units are only interleaved at $\mathcal{IO}$ steps, as depicted in Fig. 15. We call such schedules *$\mathcal{IO}$-block schedules* and below we define this notion of interleaving in our step sequence formalism.

Definition 22 ($\mathcal{IO}$-Block Schedule) *We define the predicate*

$$\begin{aligned}\mathcal{IO}sched(\theta) \quad \equiv \quad & \forall \theta' \in \Theta_S^*, \alpha, \beta \in \Theta_S.\ \theta = \theta'\alpha\beta \implies \\ & \mathcal{IO}sched(\theta'\alpha) \wedge (\alpha.s = \beta.s \vee \beta.io \vee \theta'|_{\beta.s} = \varepsilon)\end{aligned}$$

that expresses whether a step sequence θ exhibits an $\mathcal{IO}$-block schedule.

In an $\mathcal{IO}$-block schedule a block of some unit p may be executed if it starts with an $\mathcal{IO}$ operation or if p was never scheduled before. The latter condition is necessary because not every unit might start executing with an $\mathcal{IO}$ step.

In what follows we show that this kind of coarse scheduling is just a special case of our reduction theorem. Apart from the requirement that units may start with an ordinary step, the only difference between the definitions of $\mathcal{IO}sched$ and $\mathcal{IP}sched$ is the fact that the $\mathcal{IO}$ predicate is used instead of $\mathcal{IP}$. Hence one is tempted to just set the $\mathcal{IP}$ predicate equal to the $\mathcal{IO}$ predicate. However we need to care about the first step of the unit which should be an interleaving-point but may not be an $\mathcal{IO}$-point. For every *Cosmos* machine S we propose the following extension yielding the extended model S'.

- $S'.\mathcal{U}$ - the unit state $u \in S'.\mathcal{U}$ contains all components of $S.\mathcal{U}$ but in addition also a boolean flag $u.init \in \mathbb{B}$ denoting that the unit has not been stepped yet. Initially we should have $u.init = 1$. If a component with the name $init$ already exists, we choose a different, unique name for the flag.

- $S'.\delta$ - the transition function is defined as before, but the semantics of the new component are defined by $u'.init = 0$ with $u' = S'.\delta(u, m, in)$, i.e., in the first step the flag is turned off and never enabled again.

The $\mathcal{IP}$ predicate can then be defined in terms of the $\mathcal{IO}$ predicate and the $init$ flag.

$$S'.\mathcal{IP}(u, m, in) = S'.\mathcal{IO}(u, m, in) \vee u.init$$

With this setting we see directly that the following statement holds.

Corollary 1 (Correct Instantiation for $\mathcal{IO}$-Block Schedules) *Given is a machine state $M \in \mathbb{M}_{S'}$ of a* Cosmos *machine where S' has been instantiated as outlined above. If the $init$ flags of all units are enabled in M, then every $\mathcal{IP}$ schedule computation out of M is also an $\mathcal{IO}$-block schedule computation.*

$$\forall M, \theta.\ (\forall p \in \mathbb{N}_{nu}.\ M.u(p).init) \wedge comp(M, \theta) \wedge \mathcal{IP}sched(\theta) \implies \mathcal{IO}sched(\theta)$$

Therefore by the verification of all $\mathcal{IO}$-block schedules we also verify all interleaving-point schedules. Since all $\mathcal{IO}$ steps start in interleaving-points and the first step of all processors is an interleaving-point, the $\mathcal{IOIP}$ condition holds trivially on all computations. Then Theorem 1 enables property transfer from coarse $\mathcal{IO}$ block schedules to schedules with arbitrary interleaving of processor steps. Thus we have shown that coarse scheduling is justified by the $\mathcal{IP}$ schedule reduction theorem. All we had to do for achieving this, was chosing the interleaving-points properly.

This finishes our chapter on order reduction. In the next chapters we shall instantiate the *Cosmos* model on various levels of our pervasive semantic stack and introduce sequential simulation theorems between them. Finally we will use $\mathcal{IP}$ schedule reordering in order to apply these simulation theorems in the concurrent case, obtaining system-wide concurrent simulation theorems.

4 *Cosmos* Model Instantiations

In this chapter we will introduce several instantiations of our *Cosmos* model. Any such instantiation needs to refine the components of a *Cosmos* machine $S \in \mathbb{S}$ which we list again below as a reminder.

$$S = (\mathcal{A}, \mathcal{V}, \mathcal{R}, nu, \mathcal{U}, \mathcal{E}, reads, \delta, \mathcal{IO}, \mathcal{IP})$$

Moreover for every instantiation we have do discharge instantiation restriction $insta_r(S)$ on the $reads$-function which determines the $reads$-set for a step of a computation unit. We will first present a *Cosmos* machine with MIPS processor units. To keep the example concise we will use a simplified MIPS [KMP14] instruction set architecture (ISA) without caches, address translation, store buffers, and devices.

Moreover we will be concerned with higher level language semantics, since system software is usually not written in machine code. Therefore we will establish A. Shadrin's Macro Assembler (MASM) semantics [Sha12] on the MIPS platform and introduce semantics for an intermediate language of C [SS12]. For both semantics we will show that we can instantiate the *Cosmos* model accordingly, obtaining models of concurrently executing C-IL or MASM computation units.

In addition we will examine the corresponding consistency relations which tie the high level configurations to the MIPS ISA implementation in the sequential assembler and compiler correctness theorems. In the subsequent chapter we will then present our simulation theory which allows to combine the sequential correctness theorems into a concurrent one, using the $\mathcal{IP}$ schedule order reduction theorem.

4.1 MIPS ISA

A natural candidate for instantiation of units in a concurrent system are processors. In this section we present an instantiation of the *Cosmos* model with a simplified MIPS instruction set architecture based on the sequential MIPS machine model documented in [KMP14]. It is a simplified processor model since it does not contain a memory management unit, store buffers, and caches, nor does it support communication with devices or inter-processor interrupts. However it is possible to handle exceptions and external interrupts.

Building on previous work of W. J. Paul, S. Schmaltz documented an extension of the simple MIPS model with the features named above from the x86 architecture, obtaining a realistic model of a multi-core RISC (Reduced Instruction Set Computing) processor, called MIPS-86 [Sch13a]. We could represent this more elaborate model as

a *Cosmos* machine if we allowed for different unit instantiations[1]. However the aim of this chapter is to provide concise examples for instantiating *Cosmos* machines on different levels of abstractions of a concurrent system and show how to establish simulation theorems, like concurrent compiler correctness, between them. In order to illustrate our simulation theory clearly, we should choose examples that are small and not cluttered with too many technical details. Therefore we stick with the simple model from [KMP14], adding only the interrupt mechanism and the compare-and-swap instruction from [Sch13b].

Nevertheless M. Kovalev showed in the context of hypervisor verification how a detailed system-programmer's view of the x64 architecture can be reduced to a corresponding user programming model, where caches, store buffers, and the memory management unit are invisible [Kov13]. Sequential compiler correctness is then applied on the reduced level of specification, assuming an order reduction theorem like the one presented in the previous chapter. We are quite confident that such a series of reductions can also be applied on a MIPS-86 *Cosmos* machine instantiation without devices and APIC, resulting in the simplified concurrent MIPS model presented below. However, proving this statement is beyond the scope of this thesis.

4.1.1 Instruction Set

We first introduce the syntax of the instructions of our simplified MIPS ISA. Actually MIPS instructions are bit strings of length 32. However we abstract from the actual binary encoding of instructions, which can be found in [KMP14], and instead use an assembly-like representation. This will ease the integration of the instruction set into the Macro Assembler language later on.

Definition 23 (MIPS Instruction Set) *The set $\mathbb{I}_{\mathrm{MIPS}}$ contains all MIPS instructions and is a finite language described by the context-free grammar G_{MIPS}.*

$$G_{\mathrm{MIPS}} = (T_{\mathrm{MIPS}}, N_{\mathrm{MIPS}}, S_{\mathrm{MIPS}}, P_{\mathrm{MIPS}})$$

Thus we have $\mathbb{I}_{\mathrm{MIPS}} = L(G_{\mathrm{MIPS}})$. We define the terminals as

$$T_{\mathrm{MIPS}} = \{0, 1, \lrcorner\} \cup \mathit{INSTR}_{\mathrm{MIPS}}$$

where the set $\mathit{INSTR}_{\mathrm{MIPS}} = \{\mathrm{add}, \ldots, \mathrm{xori}\}$ contains the names of all MIPS instructions that we want to treat below. Non-terminals are marked by angle brackets.

$$\begin{aligned} N_{\mathrm{MIPS}} \quad = \quad & \{\langle \mathit{Instr}\rangle, \langle \mathit{rd}\rangle, \langle \mathit{rs}\rangle, \langle \mathit{rt}\rangle, \langle \mathit{imm}\rangle, \langle \mathit{iindex}\rangle, \langle \mathit{Itype}\rangle, \langle \mathit{Jtype}\rangle, \langle \mathit{Rtype}\rangle, \\ & \langle \mathit{aluI}\rangle, \langle \mathit{ls}\rangle, \langle \mathit{branch0}\rangle, \langle \mathit{branch}\rangle, \langle \mathit{aluR}\rangle\} \end{aligned}$$

[1] We would have different units for executing steps of processors, devices and advanced programmable interrupt controllers (APIC).

We set the starting symbol to $S_{\mathrm{MIPS}} = \langle Instr \rangle$. The production system P_{MIPS} is defined by the following rules:

$$
\begin{array}{rcl}
\langle Instr \rangle & \longrightarrow & \langle Itype \rangle \mid \langle Jtype \rangle \mid \langle Rtype \rangle \\
\langle rd \rangle & \longrightarrow & \mathbb{B}^5 \\
\langle rt \rangle & \longrightarrow & \mathbb{B}^5 \\
\langle rs \rangle & \longrightarrow & \mathbb{B}^5 \\
\langle imm \rangle & \longrightarrow & \mathbb{B}^{16} \\
\langle iindex \rangle & \longrightarrow & \mathbb{B}^{26} \\
\langle Itype \rangle & \longrightarrow & \langle aluI \rangle \ \langle rt \rangle \ \langle rs \rangle \ \langle imm \rangle \mid \text{lui} \ \langle rt \rangle \ \langle imm \rangle \mid \langle \text{ls} \rangle \ \langle rt \rangle \ \langle rs \rangle \ \langle imm \rangle \mid \\
 & & \langle branch \rangle \ \langle rs \rangle \ \langle rt \rangle \ \langle imm \rangle \mid \langle branch0 \rangle \ \langle rs \rangle \ \langle imm \rangle \\
\langle aluI \rangle & \longrightarrow & \text{addi} \mid \text{addui} \mid \text{slti} \mid \text{sltui} \mid \text{andi} \mid \text{ori} \mid \text{xori} \\
\langle ls \rangle & \longrightarrow & \text{lw} \mid \text{sw} \\
\langle branch \rangle & \longrightarrow & \text{beq} \mid \text{bne} \\
\langle branch0 \rangle & \longrightarrow & \text{bgez} \mid \text{bgtz} \mid \text{blez} \mid \text{bltz} \\
\langle Jtype \rangle & \longrightarrow & \text{j} \ \langle iindex \rangle \mid \text{jal} \ \langle iindex \rangle \\
\langle Rtype \rangle & \longrightarrow & \langle aluR \rangle \ \langle rd \rangle \ \langle rs \rangle \ \langle rt \rangle \mid \text{jr} \ \langle rs \rangle \mid \text{jalr} \ \langle rd \rangle \ \langle rs \rangle \ \mid \\
 & & \text{sysc} \mid \text{eret} \mid \langle mov \rangle \ \langle rd \rangle \ \langle rt \rangle \mid \text{cas} \ \langle rd \rangle \ \langle rs \rangle \ \langle rt \rangle \\
\langle aluR \rangle & \longrightarrow & \text{add} \mid \text{addu} \mid \text{sub} \mid \text{subu} \mid \text{and} \mid \text{or} \mid \text{xor} \mid \text{nor} \mid \text{slt} \mid \text{sltu} \\
\langle mov \rangle & \longrightarrow & \text{movg2s} \mid \text{movs2g}
\end{array}
$$

Note that the grammar is unambiguous because every instruction starts with a unique mnemonic. Moreover the mnemonic and parameter bit strings of instructions are separated by spaces that are part of the language. For readability we refrained from displaying them explicitly in the rules above. Thus, for any mnemonic followed by a certain number of parameter bit strings that are separated by spaces, there is at most one derivation tree wrt. G_{MIPS}.

Note also that we have introduced some redundancy in the grammar. For instance we have the three non-terminals $\langle rd \rangle$, $\langle rt \rangle$, and $\langle rs \rangle$ which all produce a 5-digit bit string. Also there different productions that produce the same pattern of bit string parameters but just differ in their possible mnemonics, e.g., the productions out of $\langle Itype \rangle$ including $\langle aluI \rangle$ and $\langle ls \rangle$. Having more non-determinals allows us to structure the instruction set definition in an intuitive way. Below, it also enables us to define instruction-based predicates easily using pattern matching on derivations.

In the grammar we see that MIPS instructions are divided into *Itype* (immediate), *Jtype* (jump), and *Rtype* (register) instructions. Register indices rd, rs, and rt are used to identify destination, source, and target registers for a given *Rtype* instruction, though not every such instruction may in fact operate on three registers, e.g. the *Rtype* system call instruction sysc does not have register parameters at all. *Itype* instructions have at most two register operands and in addition one immediate operand which is a 16-bit

constant. $Jtype$ instructions represent jumps in the control flow and have only one 26-bit immediate operand. Below we give a short explanation of the various instructions of our simplified MIPS ISA. A formal definition will follow in the next sections.

- **ALU Instructions.** These are arithmetical and logical operations on registers and immediate constants (in case of $Itype$). The operations include signed and unsigned binary addition ($\mathrm{add}, \mathrm{addi}, \mathrm{addu}, \mathrm{addui}$), subtraction ($\mathrm{sub}, \mathrm{subu}$), logical AND, OR, XOR, and NOR ($\mathrm{and}, \mathrm{andi}, \mathrm{or}, \mathrm{ori}, \mathrm{xor}, \mathrm{xori}, \mathrm{nor}$), as well as test-and-set instructions ($\mathrm{slt}, \mathrm{slti}, \mathrm{sltu}, \mathrm{sltui}$), and the lui instruction which loads a 16-bit immediate constant in the upper half of a 32-bit register. The indices rs, rt, and rd (in case of $Rtype$) determine the operands for the computation. For example $Itype$ instruction $\mathrm{addi}\ rt\ rs\ imm$ has the effect that the sign-extended immediate constant imm is added to the content of register rs. The 32-bit result is then stored in register rt. For $Rtype$ instruction $\mathrm{or}\ rd\ rs\ rt$ the binary OR of the contents of registers rs and rt is stored in register rd. Concerning the test-and-set operations the signed or unsigned binary value of register rs is compared with either the value of rt (in case of $Rtype$) or the sign-extended immediate constant (for $Itype$). If the first is less than the second, rd (rt for $Itype$) is set to 1, otherwise it is cleared to 0.

- **Branch Instructions.** Branches are used to change the control-flow of a program on the ISA level. Depending on a certain condition, program execution returns either at the next instruction or it jumps to a location which is specified using the 16-bit immediate constant. The condition can be stated as a comparison of the rs register content with zero, or as a test whether two registers rs and rt contain the same data. For instance $\mathrm{bltz}\ rs\ imm$ branches to a location with offset $imm00$ to the current one if register rs contains a negative value. On the other hand $\mathrm{beq}\ rs\ rt\ imm$ results in a similar control-flow jump, when registers rs and rt have equal content.

- **Jump Instructions.** $Jtype$ jumps are unconditional branches to an absolute address that is computed using the 26-bit immediate address $iindex$. Instruction j is the plain jump while jal (Jump and Link) additionally stores in a specific register the memory location of the subsequent instruction in the program code.

  There are also similar $Rtype$ jump instructions which use the content of register rs as an absolute target address for the jump. Jump and Link Register (jalr) allows to specify register rd in which the address of the subsequent instruction will be stored. In addition there is the sysc instruction which is used to invoke a system call handler in an operating system.

- **Memory Instructions.** In a RISC architecture there are dedicated instructions for transferring data between the processor's registers and memory. In our simplified MIPS ISA we have a load instructions $\mathrm{lw}\ rt\ rs\ imm$ (Load Word) and a store instruction $\mathrm{sw}\ rt\ rs\ imm$ (Store Word), where a *word* is a string of 32 bis. That means that we do not consider instructions here that update memory at the half-word or

byte granularity. The *effective address* which is read or written by the instruction is determined by adding the sign-extended immediate constant imm to the content of register rs. A value loaded from memory is then stored in register rs, while for "Store Word" instructions memory is updated with the content of rt.

In addition an atomic compare-and-swap operation can be performed using $Rtype$ instruction $\mathrm{cas}\ rd\ rs\ rt$. There the memory content x at the address in register rs is updated with the value in register rt in case x is equal to the value of rd.

- **Coprocessor Instructions.** These are instructions that are needed especially for interrupt handling. Instruction eret (Exception Return) is used to return from an interrupt handler to the interrupted program, reversing the effects of the jump to the interrupt service routine (JISR). We use instructions $\mathrm{movg2s}$ and $\mathrm{movs2g}$ to move data between the normal general purpose registers of the MIPS processor and certain special purpose registers. In the simplified MIPS ISA presented here we only treat the special purpose registers necessary for interrupt handling.

We assume a function $decode : \mathbb{B}^{32} \to \mathbb{I}_{\mathrm{MIPS}} \cup \{\bot\}$ which transforms any 32-bit binary instruction representation into the assembly-like instruction representation introduced above, using the tables and definitions from [KMP14]. For invalid encodings the function returns $\bot$. Moreover for convenience we define predicates to distinguish what kind of instructions are represented by some $I \in \mathbb{I}_{\mathrm{MIPS}}$.

- $Itype(I) \stackrel{def}{\equiv} \langle Itype \rangle \longrightarrow^*_{G_{\mathrm{MIPS}}} I$ — I is an $Itype$ instruction.
- $Jtype(I) \stackrel{def}{\equiv} \langle Jtype \rangle \longrightarrow^*_{G_{\mathrm{MIPS}}} I$ — I is a $Jtype$ instruction.
- $Rtype(I) \stackrel{def}{\equiv} \langle Rtype \rangle \longrightarrow^*_{G_{\mathrm{MIPS}}} I$ — I is an $Rtype$ instruction.
- $mne(I) \stackrel{def}{\equiv}$ x such that $I \in \{\mathrm{x}, \mathrm{x}\_y\}$ where $\mathrm{x} \in INSTR$ and $y \in \{0, 1, \_\}^*$ — returns the mnemonic of instruction I.
- $\forall \mathrm{x} \in INSTR.\ \mathrm{x}(I) \stackrel{def}{\equiv} (mne(I) = \mathrm{x})$ — e.g., $\mathrm{lw}(I)$ is true iff I is a lw instruction.
- $\forall x \in \{aluI, aluR, mov, branch0\}.\ x(I) \stackrel{def}{\equiv} \langle x \rangle \longrightarrow^*_{G_{\mathrm{MIPS}}} mne(I)$ — e.g., $mov(I)$ means that I is a coprocessor instruction for accessing special purpose registers.
- $alu(I) \stackrel{def}{\equiv} aluI(I) \vee \mathrm{lui}(I) \vee aluR(I)$ — I is an ALU instruction.
- $branch(I) \stackrel{def}{\equiv} branch0(I) \vee \mathrm{beq}(I) \vee \mathrm{bne}(I)$ — I is a branch instruction.
- $jump(I) \stackrel{def}{\equiv} Jtype(I) \vee \mathrm{jr}(I) \vee \mathrm{jalr}(I)$ — I is a jump instruction.
- $ctrl(I) \stackrel{def}{\equiv} jump(I) \vee branch(I) \vee \mathrm{eret}(I) \vee \mathrm{sysc}(I)$ — I is a control-flow instruction.

- $mem(I) \stackrel{def}{\equiv} \text{ls}(I) \vee \text{lw}(I) \vee \text{cas}(I)$ — I is an instruction that accesses memory.

We define further functions to parse the instruction parameters following the mnemonic. Below let imm, $iindex$, rd, rs, and rt be bit strings and $x, y \in \{0, 1, \_\}^*$ and $mne \in INSTR_{\text{MIPS}}$. Remember that the parameters are separated by spaces, therefore we can use pattern matching to filter out the desired parameters from a given derivation.

- immediate constant:

$$imm(I) \stackrel{def}{\equiv} \begin{cases} imm[15:0] & : \quad Itype(I) \wedge I = mne\ x\ y\ imm \\ & \quad \vee\ branch0(I) \wedge I = mne\ x\ imm \\ iindex[25:0] & : \quad Jtype(I) \wedge I = mne\ iindex \\ \bot & : \quad \text{otherwise} \end{cases}$$

- source register:

$$rs(I) \stackrel{def}{\equiv} \begin{cases} rs[4:0] & : \quad (alu(I) \vee mem(I)) \wedge I = mne\ x\ rs\ y \vee I = \text{jr}\ rs \\ & \quad \vee\ branch(I) \wedge I = mne\ rs\ x \vee I = \text{jalr}\ rd\ rs \\ \bot & : \quad \text{otherwise} \end{cases}$$

- target register:

$$rt(I) \stackrel{def}{\equiv} \begin{cases} rt[4:0] & : \quad (Itype(I) \wedge /branch(I)) \wedge I = mne\ rt\ x \\ & \quad \vee\ (\text{beq}(I) \vee \text{bne}(I)) \wedge I = mne\ rs\ rt\ imm \\ & \quad \vee\ (aluR(I) \vee \text{cas}(I)) \wedge I = mne\ rd\ rs\ rt \\ & \quad \vee\ mov(I) \wedge I = mne\ rd\ rt \\ \bot & : \quad \text{otherwise} \end{cases}$$

- destination register (only for *Rtype* instructions):

$$rd(I) \stackrel{def}{\equiv} \begin{cases} rd[4:0] & : \quad (aluR(I) \vee mov(I) \vee \text{jalr}(I) \vee \text{cas}(I)) \wedge I = mne\ rd\ x \\ \bot & : \quad \text{otherwise} \end{cases}$$

4.1.2 Configuration

A simplified MIPS configuration consists of processor registers and the main memory. This distinction is also reflected in the formal definition.

Definition 24 (MIPS Configuration) *The configuration h of a sequential MIPS processor contains a core and a memory component.*

$$h = (c, m) \in \mathbb{H}_{\text{MIPS}}$$

Here $m : \mathbb{B}^{32} \to \mathbb{B}^8$ represents the byte-addressable main memory and $c \in \mathbb{K}_{\text{MIPS}}$ represents the processor core with the following components.

$\langle i \rangle$	name for i	description
0	sr	status register (contains masks to enable/disable maskable interrupts)
1	esr	exception sr (saves sr in case of interrupt)
2	eca	exception cause register (saves cause for interruption)
3	epc	exception pc (address to return to after interrupt handling)
4	edata	exception data (external interrupt identifier)
7	mode	mode register (0^{32} for system mode, $0^{31}1$ for user mode)
8	emode	exception mode register (saves mode in case of interrupt)

Table 4.1: MIPS Special Purpose Registers, where $i \in \mathbb{B}^5$.

- $c.pc \in \mathbb{B}^{32}$ — *the program counter (PC)*
- $c.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$ — *the general purpose register file (GPR) consisting of 32 registers that are 32 bits wide and can be addressed with a 5-bit index.*
- $c.spr : \mathbb{B}^5 \to \mathbb{B}^{32}$ — *the special purpose register file (GPR) consisting of 32 registers, however only six of the 32 registers are used in the simplified MIPS model that are 32 bits wide and can be addressed with a 5-bit index.*

The program counter contains the memory address where is stored the binary encoding of the instruction to be executed next. We use the general purpose registers to store intermediate results of computations. One register can hold a 32-bit binary value which can be interpreted as signed and unsigned binary numbers as well as memory addresses. The special purpose registers can only be accessed by the mov instructions, sysc, and eret. Here we use them only for interrupt handling. The dedicated special purpose registers are depicted in Table 4.1. It is taken from [Sch13b] but we ommitted the support for address translation here to keep the model simple.

4.1.3 Semantics

In this section we will define a simplified MIPS semantics by introducing a transition function $\delta_{\mathrm{MIPS}} : \mathbb{H}_{\mathrm{MIPS}} \times \mathbb{B}^{256} \to \mathbb{H}_{\mathrm{MIPS}}$, which performs a computation step on a given MIPS configuration, taking into account a 256-bit external event vector (eev), resulting in a new MIPS configuration. The transition function basically has two cases, where in the first case the next instruction according to the program counter is executed, and in the other case a jump to the interrupt service routine (JISR) is performed after an interrupt was signalled via the external event vector. In what follows we introduce additional notation in order to define δ_{MIPS} formally.

Auxiliary Definitions and Intermediate Results

As mentioned above the instruction to be executed next in a given MIPS configuration h is fetched from memory using the program counter $h.c.pc$. We use notation $I(h)$ to denote this next instruction.

$$I(h) \stackrel{def}{\equiv} decode(h.m_4(h.c.pc))$$

Since instructions are encoded as 4-byte-wide binary values, the encoding of the current instruction is stored in memory in the four consecutive bytes starting at the program counter. There is a software condition that memory is only accessed in a *word-aligned* way, this means that memory is divided into 2^{30} consecutive words starting at byte address 0^{32} and one must never cross a word boundary in a single memory access. This can be guaranteed if the least two significand bits of an address in any memory access are always zero. In the case of instruction fetch this means:

$$\forall h \in \mathbb{H}_{\mathrm{MIPS}}.\ h.c.pc[1:0] = 00$$

A violation of this software condition results in a *misalignment* interrupt. However if the access is aligned and the fetched word encodes in deed a valid MIPS instruction ($I(h) \neq \perp$), by $I(h)$ we obtain the assembly-like representation of the instruction to be executed next. We only consider this case in the following defintions. Then we can extend the predicates on instructions defined above to predicates about the the next instruction depending on the MIPS configuration. For example predicate $mem(h)$ denotes that the next instruction accesses memory.

$$\forall p : \mathbb{I}_{\mathrm{MIPS}} \to \mathbb{B}, h \in \mathbb{H}_{\mathrm{MIPS}}.\quad p(h) \stackrel{def}{\equiv} p(I(h))$$

The same principle applies to the shorthands defined above for accessing the various parameters of a MIPS instruction. With $rd(h)$, $rs(h)$, and $rt(h)$ we can obtain the fields of the current instruction to address the GPR in order to load operands and store results of instruction execution. Moreover some of the following definitions will only depend on the current MIPS processor core configuration c and the current instruction I. We allow such functions $f(c, I)$ mapping to some image X to be applied on a complete MIPS configuration $h \in \mathbb{H}_{\mathrm{MIPS}}$ in the obvious way.

$$\forall f : \mathbb{K}_{\mathrm{MIPS}} \times \mathbb{I}_{\mathrm{MIPS}} \to X.\quad f(h) \stackrel{def}{\equiv} f(h.c, I(h))$$

The 16-bit and 26-bit immediate constants have various uses, however, since our MIPS architecture is based on 32-bit words we need to convert the constants to 32-bit format using sign-extension (for signed numbers) and zero-extension (for unsigned numbers). To this end we introduce the following two functions.

$$sxtimm_m^n(a) \stackrel{def}{\equiv} a_{m-1}^{n-m} a[m-1:0] \qquad zxtimm_m^n(a) \stackrel{def}{\equiv} 0^{n-m} a[m-1:0]$$

The n-bit sign-extension $sxtimm_m^n(a)$ of an m-bit number $a[m-1:0]$ extends the bit string to length n by duplicating the most significand bit of a. This operation preserves the signed two's-complement value of a. Similarly the n-bit zero-extension $zxtimm_m^n(a)$ preserves the unsigned binary value of a. For logical operations of *Itype* instructions we use the zero-extended immediate constant, for arithmetic operations the sign-extended version is used. In general, the extended immediate constant of the current instruction is defined by:

$$xtimm(I) \stackrel{def}{\equiv} \begin{cases} zxtimm_{16}^{32}(imm(I)) & : \quad \text{andi}(I) \vee \text{ori}(I) \vee \text{xori}(I) \\ sxtimm_{16}^{32}(imm(I)) & : \quad \text{otherwise} \end{cases}$$

Note that this definition is only well-defined if $/Rtype(h)$ holds, however we omit the explicit case distinction here and assume the convention that every function evaluates to $\perp$ in case its definition is not applicable for a given argument.

For arithmetic, logical, test-and-set, as well as branch instructions we usually have a left and a right operand that are used to compute the result of the computation. Generally we use the value in the GPR specified by rs as the left operand.

$$lop(c, I) \stackrel{def}{\equiv} c.gpr(rs(I))$$

The right operand is either the content of register rt (in case of *Rtype* instructions), the left-shifted 16-bit immediate constant (for lui instruction), or the extended immediate constant.

$$rop(c, I) \stackrel{def}{\equiv} \begin{cases} c.gpr(rt(I)) & : \quad Rtype(I) \\ imm(I) \circ 0^{16} & : \quad \text{lui}(I) \\ xtimm(I) & : \quad \text{otherwise} \end{cases}$$

We define the result of an ALU instruction by the following function $ares$, which depends upon left operand $lop \in \mathbb{B}^{32}$, right operand $rop \in \mathbb{B}^{32}$, and the mnemonic $mne \in INSTR_{\text{MIPS}}$ of the instruction to be executed.

$$ares(rop, lop, mne) \stackrel{def}{\equiv} \begin{cases} lop +_{32} rop & : \quad mne \in \{\text{add}, \text{addi}, \text{addu}, \text{addui}\} \\ lop -_{32} rop & : \quad mne \in \{\text{sub}, \text{subu}\} \\ lop \wedge rop & : \quad mne \in \{\text{and}, \text{andi}\} \\ lop \vee rop & : \quad mne \in \{\text{or}, \text{ori}\} \\ lop \oplus rop & : \quad mne \in \{\text{xor}, \text{xori}\} \\ lop \mathbin{\overline{\vee}} rop & : \quad mne = \text{nor} \\ rop & : \quad mne = \text{lui} \\ 0^{31}1 & : \quad mne \in \{\text{slt}, \text{slti}\} \wedge [lop] < [rop] \\ & \quad \vee\ mne \in \{\text{sltu}, \text{sltui}\} \wedge \langle lop \rangle < \langle rop \rangle \\ 0^{32} & : \quad \text{otherwise} \end{cases}$$

We can then easily define the ALU computation result for a given configuration c and MIPS instruction I.

$$ares(c, I) \stackrel{def}{\equiv} ares(rop(c, I), lop(c, I), mne(I))$$

The result of an operation is stored in either register rd or rt.

$$rdes(I) \stackrel{def}{\equiv} \begin{cases} rd(I) & : \quad Rtype(I) \\ rt(I) & : \quad \text{otherwise} \end{cases}$$

In case of memory accesses an address computation needs to be performed. The *effective address* (ea) for the memory access is determined for Store Word and Load Word instructions by adding the sign-extended immediate constant to the value stored in register rs. For the $Rtype$ instruction cas no immediate offset is available and only the value in rs is used.

$$ea(c, I) \stackrel{def}{\equiv} \begin{cases} c.gpr(rs(I)) +_{32} xtimm(I) & : \quad Itype(I) \\ c.gpr(rs(I)) & : \quad \text{otherwise} \end{cases}$$

We have the software condition that effective addresses are word-aligned, i.e.:

$$\forall h \in \mathbb{H}_{\mathrm{MIPS}}.\ ea(h)[1:0] = 00$$

For branch instructions we need to evaluate a branch condition by comparing the value in rs with the right branch operator $brop$, which is either 0 or the value stored in rt.

$$brop(c, I) \stackrel{def}{\equiv} \begin{cases} 0^{32} & : \quad branch0(I) \\ c.gpr(rt(I)) & : \quad \text{otherwise} \end{cases}$$

The result of the evaluation is then defined as follows for left operand lop, right branch operand $brop$, and instruction mnemonic mne.

$$btaken(lop, brop, mne) \stackrel{def}{\equiv} \begin{cases} [lop] < [brop] & : \quad mne = \mathrm{bltz} \\ [lop] \leq [brop] & : \quad mne = \mathrm{blez} \\ [lop] \geq [brop] & : \quad mne = \mathrm{bgez} \\ [lop] > [brop] & : \quad mne = \mathrm{bgtz} \\ lop \neq brop & : \quad mne = \mathrm{bne} \\ lop = brop & : \quad \text{otherwise} \end{cases}$$

Naturally we define a corresponding predicate for a given MIPS configuration.

$$btaken(c, I) \stackrel{def}{\equiv} btaken(lop(c, I), brop(c, I), mne(I))$$

Regular Instruction Execution

With the auxiliary defintions from above we can now define the semantics of regular instruction execution. The latter is defined by the absence of interrupts and that the current instruction is not eret. These cases shall be treated separately.

Definition 25 (Regular Instruction Semantics) *We define the transition function for regular instruction execution as a mapping* $\delta_{instr} : \mathbb{H}_{\text{MIPS}} \times \mathbb{I}_{\text{MIPS}} \to \mathbb{H}_{\text{MIPS}}$*. For a given MIPS instruction* I *an old MIPS configuration* h *is mapped to a new configuration* $h' = \delta_{instr}(h, I)$ *according to the following cases distinction.*

- *GPR — We store the results of ALU instructions and values loaded from memory. In case of jump-and-link instructions we save the next sequential program counter. For* movs2g *we load the specified SPR register value.*

$$h'.c.gpr(r) \equiv \begin{cases} ares(h.c, I) & : \quad alu(I) \wedge r = rdes(I) \\ h.m_4(ea(h.c, I)) & : \quad (\text{lw}(I) \vee \text{cas}(I)) \wedge r = rdes(I) \\ h.c.pc +_{32} 4_{32} & : \quad \text{jr}(I) \wedge r = 1^5 \vee \text{jalr}(I) \wedge r = rdes(I) \\ h.c.spr(rd(I)) & : \quad \text{movs2g}(I) \wedge r = rt(I) \\ h.c.gpr(r) & : \quad \text{otherwise} \end{cases}$$

- *SPR — Special purpose registers can only be updated by the* movg2s *instruction.*

$$h'.c.spr(r) \equiv \begin{cases} h.c.gpr(rt(I)) & : \quad \text{movg2s}(I) \wedge r = rd(I) \\ h.c.spr(r) & : \quad \text{otherwise} \end{cases}$$

- *Memory — Only instructions Load Word and Compare-and-Swap may modify memory. Let*

$$swap(h, I) \stackrel{def}{\equiv} \text{cas}(I) \wedge h.m_4(ea(h.c, I)) = h.c.gpr(rd(I))$$

denote that the comparison in a cas *instruction was evaluated to true, thus allowing the atomic memory update. Using alignment of effective addresses we define for all* $a \in \mathbb{B}^{30}$*:*

$$h'.m_4(a00) = \begin{cases} h.c.gpr(rt(I)) & : \quad (\text{sw}(I) \vee swap(h, I)) \wedge a00 = ea(h.c, I) \\ h.m_4(a00) & : \quad \text{otherwise} \end{cases}$$

- *PC — The program counter can be manipulated by branch and jump instructions. For non-control-flow instructions it is simply incremented by* 4 *in each execution step.*

$$h'.c.pc = \begin{cases} (h.c.pc +_{32} 4_{32})[31:28] \circ imm(I) \circ 00 & : \quad \text{j}(I) \vee \text{jal}(I) \\ h.c.gpr(rs(I)) & : \quad \text{jr}(I) \vee \text{jalr}(I) \\ h.c.pc +_{32} sxtimm_{18}^{32}(imm(I) \circ 00) & : \quad \text{branch}(I) \wedge \text{btaken}(h.c, I) \\ h.c.pc +_{32} 4_{32} & : \quad \text{otherwise} \end{cases}$$

Recall above that for Jtype instructions the immediate constant is 26 bits wide.

Thus we have defined semantics for the cases that are unrelated to interrupt handling.

interrupt level	shorthand	int/ext	type	maskable	description
0	reset	external	abort	no	reset signal
1	dev	external	repeat	yes	device interrupt
2	ill	internal	abort	no	illegal instruction
3	mal	internal	abort	no	misalignment
6	sysc	internal	continue	no	system call
7	ovf	internal	continue	yes	overflow

Table 4.2: MIPS Interrupt Types and Priority.

Interrupt Semantics

Instruction execution can be interrupted because of internal or external causes. For example an internal interrupt (also called *exception*) might stem from a programming error, e.g., accessing memory with misaligned effective address. External interrupts are triggered by external signals, e.g., *reset*, that are collected in the 256-bit external event vector eev that is given as an input parameter to δ_{MIPS}. Some interrupts can be masked, meaning that they will be ignored by the MIPS processor, resuming regular instruction Moreover interrupts have different resume types, i.e., they differ in the way execution resumes after the interrupt is handled (after executing eret). We might either repeat the interrupted instruction or continue with the next instruction. The latter implies that execution of the interrupted instruction was completed before the interrupt handler was called. In case a fatal exception occured instruction execution might simply be aborted.

In our simplified MIPS architecture we support the five interrupts which are listed in Table 4.2 omitting the interrupts from [Sch13b] related to address translation. We list them below from highest to lowest priority (interrupt level).

- reset — the *reset* signal $eev[0]$ is used to bring a processor in its initial state. For computations we usually assume that reset is active in the first step and then never again.

- dev — the device interrupt is triggered by signals from the environment of the MIPS processor. Such signals can represent requests from devices but also timer alarms. Modern architectures contain advance programmable interrupt controllers (APICs) which send and deliver external interrupts, we however chose not to give an APIC model here but simply rely on the 256-bit external event vector as an input to our MIPS model. External device interrupts are maskable and any interrupted instruction is repeated after handling.

- ill — if the instruction word fetched from memory according to the program counter could not be decoded, i.e. it was in an invalid instruction format, an

illegal instruction exception is triggered. Moreover depending on the least significand bit of the $mode$ register in the SPR we are either in system or user mode.

$$user(c) \stackrel{def}{\equiv} (c.spr(\text{mode})[0] = 1)$$

If a program tries to access the special purpose registers while the processor is in user mode, an illegal instruction exception is raised, too.

- mal — a misaligment exception occurs if memory is accessed using a misaligned program counter or effective address.
- sysc — in operating systems user programs can use the system call instruction to request certain kernel functionality. Since such interrupts are called for explicitely, it would not make sense to mask them. Also they should not be repeated after return from the system call handler. Therefore they have resume type *continue*.
- ovf — an overflow exception is caused if the result of a signed arithmetic operation excedes the range of values that can be represented by 32-bit two's-complement numbers. That is, the result is smaller than -2^{31} or greater equal 2^{31}. Unsigned operations by design do not cause overflows. Furthermore overflow interrupts can be masked and the instruction producing the overflow is first completed before the overflow handler is called.

Below we define a 32-bit vector that records the causes of interrupts given a decoded instruction $I \in \mathbb{I}_{\text{MIPS}} \cup \{\perp\}$. For any predicate $p : \mathbb{I}_{\text{MIPS}} \to \mathbb{B}$ we set $p(\perp) \equiv 0$.

$$ca(c, I, eev)[j] = \begin{cases} eev[0] & : \quad j = 0 \\ \bigvee_{i=1}^{255} eev[i] & : \quad j = 1 \\ I = \perp \vee user(c) \wedge mov(I) & : \quad j = 2 \\ \{c.pc[1:0], ea(c, I)[1:0]\} \not\subseteq \{00, \perp\} & : \quad j = 3 \\ \text{sysc}(I) & : \quad j = 6 \\ [lop(c, I)] + [rop(c, I)] \notin [-2^{31} : 2^{31}) & : \quad j = 7 \wedge (\text{add}(I) \vee \text{addi}(I)) \\ [lop(c, I)] - [rop(c, I)] \notin [-2^{31} : 2^{31}) & : \quad j = 7 \wedge \text{sub}(I) \\ 0 & : \quad \text{otherwise} \end{cases}$$

To obtain the remaining interrupts after masking we AND the maskable cause bits with the corresponding mask bits from the status register. In our case only the device interrupt and overflows can be masked.

$$\forall j \in [0:31].\ mca(c, I, eev)[j] = \begin{cases} ca(c, I, eev)[j] \wedge c.spr(\text{sr})[j] & : \quad j \in \{1, 7\} \\ ca(c, I, eev)[j] & : \quad \text{otherwise} \end{cases}$$

In case of an unmasked interrupt occurring, we jump to the interrupt service routine (JISR). This condition is denoted by the following predicate.

$$jisr(c, I, eev) \stackrel{def}{\equiv} \bigvee_{i=0}^{31} mca(c, I, eev)[i]$$

Then the interrupt level il is determined by the active interrupt with highest priority.

$$il(c, I, eev) \stackrel{def}{\equiv} \min\{i \in [0:31] \mid mca(c, I, eev)[i] = 1\}$$

Therefore we have an interrupt with resume type *continue* if the highest priority interrupt is system call or overflow. Again we allow to apply the functions defined above for arguments $(h, eev) \in \mathbb{H}_{\text{MIPS}} \times \mathbb{B}^{256}$ by replacing c with $h.c$ and I with $I(h)$. Now we can define the effect of the jump to interrupt service routine transition.

Definition 26 (JISR transition) *The function $\delta_{jisr} : \mathbb{H}_{\text{MIPS}} \times \mathbb{B}^{256} \to \mathbb{H}_{\text{MIPS}}$ computes the effect of the jump to the interrupt service routine of a MIPS processor. Let $h' = \delta_{jisr}(h, eev)$ then we have for the different components:*

- *SPR — we mask all maskable interrupts by setting the status register to zero, we save the old status register, the masked cause and the current or next program counter according to the resume type of the interrupt. Moreover for device interrupts we provide the binary encoding of the lowest index of the active external event signals in eev. We switch to system mode and save the old processor mode in the* emode *register.*

$$h'.c.spr(r) = \begin{cases} 0^{32} & : \quad r = \text{sr} \\ h.c.spr(\text{sr}) & : \quad r = \text{esr} \\ mca(h, eev) & : \quad r = \text{eca} \\ h.c.pc + 4_{32} & : \quad r = \text{epc} \wedge il(h, eev) \in \{6, 7\} \\ h.c.pc & : \quad r = \text{epc} \wedge il(h, eev) \notin \{6, 7\} \\ bin_{32}(\min\{i \mid eev[i] = 1\}) & : \quad r = \text{edata} \wedge \text{il}(h, eev) = 1 \\ 0^{32} & : \quad r = \text{mode} \\ h.c.spr(\text{mode}) & : \quad r = \text{emode} \\ h.c.spr(r) & : \quad \text{otherwise} \end{cases}$$

- *PC — we jump to the interrupt service routine which by design is located at address null.*

$$h'.c.pc = 0^{32}$$

- *GPR — the general purpose registers are unchanged unless we have an overflow exception. In this case we have to complete the arithmetic operation first and commit the results to the GPR before jumping to the interrupt service routine.*

$$h'.c.gpr = \begin{cases} \delta_{instr}(h).c.gpr & : \quad il(h, eev) = 7 \\ h.c.gpr & : \quad \text{otherwise} \end{cases}$$

Although system calls have also resume type continue *we do not have to consider them here, since they do not affect the general purpose registers.*

- *Memory — Since* continue*-type interrupts do not occur for instructions that can update memory and interrupts do not modify memory themselves, the memory is unchanged by the interrupt transition.*

$$h'.m = h.m$$

After the JISR transition, execution continues in the interrupt handler. Control flow can be redirected to the point in the program where the interruption occured using the eret instruction.

Definition 27 (Effect of eret**)** *We define function* $\delta_{eret} : \mathbb{H}_{\mathrm{MIPS}} \to \mathbb{H}_{\mathrm{MIPS}}$ *which encodes the effect of the* eret *instruction as follows. With* $h' = \delta_{eret}(h)$ *we have:*

- *PC — the program counter is restored from* epc *register.*

$$h'.c.pc = h.c.spr(\mathrm{epc})$$

- *SPR – we restore the saved status register and processor mode.*

$$h'.c.spr(r) = \begin{cases} h.c.spr(\mathrm{esr}) & : \quad r = \mathrm{sr} \\ h.c.spr(\mathrm{emode}) & : \quad r = \mathrm{mode} \\ h.c.spr(r) & : \quad \mathrm{otherwise} \end{cases}$$

- *GPR and Memory — these components are not affected by* eret*.*

$$h'.c.gpr = h.c.gpr \qquad h'.m = h.m$$

This finishes the definition of interrupt semantics

Overall Transition Function

We use the transition functions defined above in order to obtain the defintion of the overall MIPS transition function δ_{MIPS}. There we make a case distinction between regular instruction execution, jump to interrupt service routine and execution of eret.

$$\delta_{\mathrm{MIPS}}(h, eev) \stackrel{def}{=} \begin{cases} \delta_{jisr}(h, eev) & : \quad jisr(h, eev) \\ \delta_{eret}(h) & : \quad /jisr(h, eev) \wedge \mathrm{eret}(h) \\ \delta_{instr}(h, I(h)) & : \quad \mathrm{otherwise} \end{cases}$$

4.1.4 *Cosmos* Machine Instantiation

Now we can define an instantiation $S^n_{\mathrm{MIPS}} \in \mathbb{S}$ which is a *Cosmos* machine containing n MIPS computation units. The components of S^n_{MIPS} are defined as follows.

- $S^n_{\mathrm{MIPS}}.\mathcal{A} = \mathbb{B}^{32}$ and $S^n_{\mathrm{MIPS}}.\mathcal{V} = \mathbb{B}^8$ — The memory is byte-addressable and contains 2^{32} memory cells.

- $S^n_{\mathrm{MIPS}}.\mathcal{R} = A_{code}$ — We assume that all code to be executed lies in an area $A_{code} \subseteq \mathcal{A}$ and we set the read-only addresses to be identical with this area.
- $S^n_{\mathrm{MIPS}}.nu = n$ — We have n computation units.
- $S^n_{\mathrm{MIPS}}.\mathcal{U} = \mathbb{K}_{\mathrm{MIPS}}$ — Every computation unit is the core of a MIPS processor.
- $S^n_{\mathrm{MIPS}}.\mathcal{E} = \mathbb{B}^{256}$ — The 256-bit external event vector is the only input to the MIPS cores.
- $S^n_{\mathrm{MIPS}}.reads$ — Depending on the executed instructions and the interrupt level different sets of addresses are loaded from memory. For $c \in \mathbb{K}_{\mathrm{MIPS}}$ and $I \in \mathbb{I}_{\mathrm{MIPS}}$ let $F(c)$ denote the addresses loaded for instruction fetch. Set $R(c, I)$ contains the addresses read by lw and cas instructions.

$$
\begin{array}{rcl}
F(c) & \stackrel{def}{\equiv} & \{c.pc, \ldots, c.pc +_{32} 3_{32}\} \\
R(c,I) & \stackrel{def}{\equiv} & \begin{cases} \{ea(c,I), \ldots, ea(c,I) +_{32} 3_{32}\} & : \quad \mathrm{lw}(I) \vee \mathrm{cas}(I) \\ \emptyset & : \quad \text{otherwise} \end{cases}
\end{array}
$$

  Now in case of regular instruction execution the $reads$-set is determined by F and R, for external interrupts no addresses are loaded, and in case of other interrupts we only fetch the interrupted instruction.

$$
S^n_{\mathrm{MIPS}}.reads(u, m, eev) = \begin{cases} F(u) \cup R(u, I(u,m)) & : \quad /jisr(u, I(u,m), eev) \\ \emptyset & : \quad il(u, I(u,m), eev) < 2 \\ F(u) & : \quad \text{otherwise} \end{cases}
$$

  Recall that $(u, m) \in \mathbb{H}_{\mathrm{MIPS}}$, thus we can use it as an input parameter to functions depending on a MIPS processor configuration $h \in \mathbb{H}_{\mathrm{MIPS}}$.

- $S^n_{\mathrm{MIPS}}.\delta$ — We define the set of written addresses $W(c, m, eev)$. A write operation is performed if the predicate $wr(c, m, eev)$ holds.

$$
\begin{array}{rcl}
wr(c, m, eev) & \stackrel{def}{\equiv} & /jisr((c,m), eev) \wedge (\mathrm{sw}(c,m) \vee swap((c,m), I(c,m))) \\
W(c, m, eev) & \stackrel{def}{\equiv} & \begin{cases} \{ea(c,m), \ldots, ea(c,m) +_{32} 3_{32}\} & : \quad wr(c, m, eev) \\ \emptyset & : \quad \text{otherwise} \end{cases}
\end{array}
$$

  Due to the $swap$ predicate these functions depend on the content of memory m. In the *Cosmos* machine the δ-function of the computation units gets only a partial memory as an input, that is determined by the $reads$-set. However the MIPS transition function is defined for a memory that is a total function. Nevertheless we can transform any partial memory function $m : \mathbb{B}^{32} \rightharpoonup \mathbb{B}^8$ into a total one by filling in dummy values.

$$
\lceil m \rceil \stackrel{def}{\equiv} \lambda a \in \mathbb{B}^{32}. \begin{cases} 0^8 & : \quad m(a) = \perp \\ m(a) & : \quad \text{otherwise} \end{cases}
$$

Then with $(u', m') = \delta_{\text{MIPS}}((u, \lceil m \rceil), eev)$ we can define the transition function for MIPS computation units which returns the same new core configuration and the updated part of memory.

$$S^n_{\text{MIPS}}.\delta(u, m, eev) = (u', m'|_{W(u,\lceil m \rceil, eev)})$$

In the *Cosmos* machine the δ function is provided a partial memory spanning exactly the $reads$-set. If $S^n_{\text{MIPS}}.reads$ is instantiated correctly, i.e., $insta_r(S^n_{\text{MIPS}})$ holds, then the computation of δ does not depend on the inserted dummy values in $\lceil m \rceil$.

- $S^n_{\text{MIPS}}.\mathcal{IO}$ — It is a task of the verification engineer to determine the $\mathcal{IO}$ steps of a system under consideration. Thus it cannot be defined in general what are the $\mathcal{IO}$ steps on the MIPS ISA level. The choice of the $\mathcal{IO}$ steps is depending on the verification methodology and the program to be verified. Below we consider the two scenarios that were already introduced in Sect. 1.1.

  1. We verify a concurrent algorithm on the MIPS ISA level that was written in MIPS assembler. All synchronization between the concurrent processors is implemented via atomic Compare-and-Swap instructions. In this case the $\mathcal{IO}$ steps are exactly the uninterrupted cas instruction executions.

     $$S^n_{\text{MIPS}}.\mathcal{IO}(u, m, eev) \equiv /jisr((u, \lceil m \rceil), eev) \wedge \text{cas}(u, \lceil m \rceil)$$

  2. Imagine we are verifying a program that was compiled down to MIPS assembler from a higher-level language like C. On the C level it is easy to determine the $\mathcal{IO}$ steps which are, for instance, accesses to volatile variables or calls to synchronization primitives that are implemented as external assembly functions (containing for example cas). In the compilation process these C statements and synchronization primitives are translated into assembly code and for a well-behaved compiler every translation of an $\mathcal{IO}$ step should contain exactly *one* shared memory access.[2]

     Knowing the code generation and placement function of the compiler, one can easily determine the addresses of these instructions that are supposed to access shared memory. We collect them in a set $A_{io} \subseteq A_{code}$. Then the definition of the $\mathcal{IO}$ steps on the ISA level is straight forward.

     $$S^n_{\text{MIPS}}.\mathcal{IO}(u, m, eev) \equiv /jisr((u, \lceil m \rceil), eev) \wedge u.pc \in A_{io}$$

- $S^n_{\text{MIPS}}.\mathcal{IP}$ — Similarly the interleaving-points are chosen by the verification engineer to determine the structure of the block schedules upon which verification methods are applied. Thus a general definition is not possible and we revisit the two examples from above.

[2] Without a shared memory access we cannot synchronize with the concurrently executing threads of the program. If one $\mathcal{IO}$ step is compiled into two or more shared memory accesses we lose atomicity, leading most probably to race conditions.

1. As we verify the concurrent program on the assembly level we want to consider as few interleavings of different processor steps as possible. Thus we would aim for a coarse $\mathcal{IO}$ block scheduling and set the interleaving-points equal to the starting points of $\mathcal{IO}$ steps and the initial states of the processors. With an $init$ flag as in Section 3.7 we have the following definition.

$$S^n_{\mathrm{MIPS}}.\mathcal{IP}(u, m, eev) \equiv S^n_{\mathrm{MIPS}}.\mathcal{IO}(u, m, eev) \vee u.init$$

2. In case of compiled code on the ISA level, we want to use the order reduction theorem to build block schedules on which the sequential compiler correctness theorem can be applied. In this way we can justify a verification of system on the C level and transfer down verified properties to the arbitrarily interleaved MIPS ISA level. To this end we need to identify the compiler consistency points on the C and ISA level, that is the locations in the C program and the compiled code where the compiler consistency relation holds between the C and ISA configurations during execution.

   Optimizing compilers do not need to guarantee compiler consistency points after every C statement but they may translate a sequence of C statements in a sequence of assembler instructions that correctly implement the C code on the ISA level. In this case the compiler consistency points are the start (and end) points of the aformentioned sequences on the C and instruction set architecture level.

   Again, for a given program and compiler, we can determine the set of MIPS ISA instruction addresses which start in compiler consistency points, i.e., the addresses of the first instruction in the compiled code for any sequence of C statements that starts in a compiler consistency point. We name this set $A_{cp} \subseteq A_{code}$. Whenever the program counter of a MIPS processor points to such an instruction, the processor is in a compiler consistency point on the ISA level, in case the compiled code is not modified. We thus set the interleaving-points to be the compiler consistency points on the level of the MIPS instruction set architecture.

$$S^n_{\mathrm{MIPS}}.\mathcal{IP}(u, m, eev) \;\equiv\; u.pc \in A_{cp}$$

   Assuming $\mathcal{IP}$ scheduling on the ISA level we can then apply the sequential compiler correctness theorem on the $\mathcal{IP}$ blocks and justify the verified view of the system where C threads are interleaved at C compiler consistency points.[3] The interleaving-point scheduling assumption can be discharged by the $\mathcal{IP}$ schedule order reduction theorem, which requires in turn that all $\mathcal{IP}$ schedules are ownership-safe and that all computations of *Cosmos* machine S^n_{MIPS} running out of a given start configuration obey the $\mathcal{IOIP}$ condition.

[3] Using the order reduction theorem again we could reduce reasoning further to coarse $\mathcal{IO}$ block schedules on the C level.

In order to show that the two requirements hold we need to consider the compiler once more. First we prove a compiler ownership transfer theorem, stating that the compilation of sequential code preserves ownership-safety. Thus from the verification of all $\mathcal{IP}$ schedules on the C level we can deduce the ownership-safety of all $\mathcal{IP}$ schedules on the compiled ISA level. For proving the $\mathcal{IOIP}$ condition it is sufficient to show that the compiler puts at most one $\mathcal{IO}$ step between two ISA-level compiler consistency points.

For most of the components the *Cosmos* model instantiation is quite obvious. Besides some technicalities about converting inputs and outputs of the δ-function, only the choice of $\mathcal{IO}$ steps and interleaving-points was non-trivial, especially in the case of compiled code running on the MIPS *Cosmos* machine. However by the program- and compiler-dependent sets A_{code}, A_{io}, and A_{cp} we were able to give precise definitions also in the latter scenario. Observe that these definitions do not depend on the ownership state, especially we did not define every access to a shared address to be automatically an $\mathcal{IO}$ step. Thus we are able to verify that shared memory is only accessed when it is expected by the verification engineer.

Moreover, note that we assume an invariant on computations of *Cosmos* machine S^n_{MIPS}, stating that A_{code} has the intended meaning, namely, that we only fetch instructions from this set of addresses.

Definition 28 (Code Region Invariant) *We define the invariant* $codeinv(H, A_{code})$ *which states that in all system states reachable from* Cosmos *machine configuration* $H \in \mathbb{C}_{S^n_{\mathrm{MIPS}}}$ *instructions are only fetched from code region* $A_{code} \subseteq \mathbb{B}^{32}$.

$$\forall \sigma, H'.\ H \overset{\sigma}{\longmapsto} H' \implies \forall p \in \mathbb{N}_{nu}.\ \{H'.u_p.pc, \ldots, H'.u_p.pc +_{32} 3_{32}\} \subseteq A_{code}$$

Since we also chose the read-only addresses set $\mathcal{R}$ to be equal to A_{code}, the ownership policy excludes self-modifying code. Thus we can assume that the compiled code lies in a fixed region of memory and cannot be altered. Therefore it is possible to use instruction addresses in order to determine $\mathcal{IO}$ steps and compiler consistency points.

Besides the code region invariant we also have the software conditions that forbid misaligned memory accesses. However in the following instantiations with the macro assembly and C-IL semantics we will demand the absence of exceptions caused by the compiled code. Thus we implicitly require that compiled code obeys alignment restrictions and therefore does not cause misalignment interrupts. Below we prove $insta_r(S^n_{\mathrm{MIPS}})$, finishing the instantiation of our MIPS ISA *Cosmos* machine.

PROOF: We need to show the following statement for MIPS processor core configuration $u \in \mathbb{K}_{\mathrm{MIPS}}$, memories $m, m' : \mathbb{B}^{32} \to \mathbb{B}^8$, external event vector $eev \in \mathbb{B}^{256}$, and $reads$-set $R = S^n_{\mathrm{MIPS}}.reads(u, m, eev)$.

$$m|_R = m'|_R \implies S^n_{\mathrm{MIPS}}.reads(u, m', eev) = R$$

Let $R' = S^n_{\mathrm{MIPS}}.reads(u, m', eev)$. The definitions of both R and R' depend on the same MIPS core configuration c, therefore processors (u, m) and (u, m') fetch an instruction

from the same set of addresses $F(u)$. We need to distinguish whether addresses are read in a step or not at all.

- $R = \emptyset$ — In this case by definition of $\mathcal{S}^n_{\mathrm{MIPS}}.reads$ an external interrupt prevented the processor from fetching an instruction. We have $il(u, I(u,m), eev) < 2$, that is either reset or a device interrupt was triggered by the external event vector.
  - reset — The interrupt level is 0. By the definition of $ca(u, I(u,m), eev)[0]$ we see that $eev[0] = 1$ and we also know that the same external even vector is used to determine R'. Thus we deduce by the definition of mca and ca:
    $$mca(u, I(u,m'), eev)[0] = ca(u, I(u,m'), eev)[0] = eev[0] = 1$$
  - dev — The interrupt level is 1. By the definition of $ca(u, I(u,m), eev)[1]$ we see that there exists some index $i \in [1:255]$ such that $eev[i] = 1$ and also $u.spr(\mathrm{sr})[1] = 1$, because the external interrupt was not masked. It follows:
    $$mca(u, I(u,m'), eev)[1] = ca(u, I(u,m'), eev)[1] \wedge u.spr(\mathrm{sr})[1] = \bigvee_{i=1}^{255} eev[i] = 1$$

  Overall we have $mca(u, I(u,m'), eev)[0] \vee mca(u, I(u,m'), eev)[1]$ and therefore:
  $$jisr(u, I(u,m'), eev) \qquad il(u, I(u,m'), eev) < 2$$
  Hence by definition of $S^n_{\mathrm{MIPS}}(u, m, eev)$ we have $R' = \emptyset = R$.

- $R \neq \emptyset$ — In the remaining two cases of the definition of $S^n_{\mathrm{MIPS}}.reads$ we have $F(u) \subseteq R$. Therefore we have $m|_{F(u)} = m'|_{F(u)}$ according to hypothesis and the same instruction is decoded by both processors because for instruction fetch they only read from $F(u) = \{u.pc, \ldots, u.pc +_{32} 3_{32}\}$.
  $$\begin{aligned} I(u,m) &= m_4(u.pc) \\ &= m(u.pc +_{32} 3_{32}) \circ \cdots \circ m(u.pc) \\ &= m|_{F(u)}(u.pc +_{32} 3_{32}) \circ \cdots \circ m|_{F(u)}(u.pc) \\ &= m'|_{F(u)}(u.pc +_{32} 3_{32}) \circ \cdots \circ m'|_{F(u)}(u.pc) \\ &= m'(u.pc +_{32} 3_{32}) \circ \cdots \circ m'(u.pc) \\ &= m'_4(u.pc) \\ &= I(u,m') \end{aligned}$$
  If we replace in the defintion of $S^n_{\mathrm{MIPS}}.reads(u, m', eev)$ all occurences of $I(u,m')$ with $I(u,m)$, then we obtain exactly the definition of $S^n_{\mathrm{MIPS}}.reads(u,m,eev)$. In particular we have:
  $$\begin{aligned} R(u, I(u,m')) &= R(u, I(u,m)) \\ jisr(u, I(u,m'), eev) &= jisr(u, I(u,m), eev) \\ il(u, I(u,m'), eev) &= il(u, I(u,m), eev) \end{aligned}$$
  Hence we conclude $R' = S^n_{\mathrm{MIPS}}.reads(u, m', eev) = S^n_{\mathrm{MIPS}}.reads(u, m, eev) = R$. □

4.2 Macro Assembly

The Macro Assembly Semantics (MASM) was developed by Andrey Shadrin [Sha12] in an effort to abstract from implementation details like the stack layout and calling conventions and provide a higher level assembly language with a dynamic stack and macros for stack operations as well as for calling of and returning from assembly functions. Furthermore the work was joined with S. Schmaltz' C-IL semantics obtaining a semantics where C-IL and MASM are executed in an interleaved fashion via external function calls. The latter will be presented in subsequent sections. First we will present the semantics of MASM and a sequential compiler consistency relation with the MIPS ISA level. Furthermore we will instantiate a MASM *Cosmos* machine and establish a concurrent compiler correctness theorem describing a simulation of the MASM *Cosmos* machine by the MIPS *Cosmos* machine.

4.2.1 MASM Syntax and Semantics

We want to stress that the MASM semantics and simulation theorem are in their entirety extracted from [Sha12]. Nevertheless we take the freedom to restructure, modify and extend definitions preserving their intention where it seems necessary in order to fit them into our framework, or where it promotes brevity and clarity. In general we base the MASM semantics on our simpified MIPS ISA model defined in the previous section.

Instructions

Macro Assembly comprises a number of macros and all ISA instructions except control-flow instructions. Thus we exclude jumps and branches but also the sysc and eret instructions. The allowed ISA instructions are defined by:

$$\mathbb{I}_{\mathrm{MIPS}}^{nocf} \stackrel{def}{\equiv} \{I \in \mathbb{I}_{\mathrm{MIPS}} \mid / ctrl(I)\}$$

In general we assume that MASM (and C-IL) programs are not interrupted. Semantics for interruptible C and MASM programs will hopefully be presented in the future (cf. Sect. 6.3.3). Formally the set of MASM instructions is denoted by the inductive type $\mathbb{I}_{\mathrm{MASM}}$ defined below, where $Names$ is the set of admissible MASM procedure names.

- $\forall r \in \mathbb{B}^5.\ \texttt{push}\ r \in \mathbb{I}_{\mathrm{MASM}} \wedge \texttt{pop}\ r \in \mathbb{I}_{\mathrm{MASM}}$ - push and pop instructions which transfer data between general purpose registers and the stack
- $\forall r \in \mathbb{B}^5, i \in \mathbb{N}.\ \texttt{lparam}\ r\ i \in \mathbb{I}_{\mathrm{MASM}} \wedge \texttt{sparam}\ r\ i \in \mathbb{I}_{\mathrm{MASM}}$ - instructions to store procedure parameters in registers or on the stack (useful for saving register space)
- $\forall P \in Names.\ \texttt{call}\ P \in \mathbb{I}_{\mathrm{MASM}}$ - calling a MASM procedure P
- $\texttt{ret} \in \mathbb{I}_{\mathrm{MASM}}$ - returning from a MASM procedure
- $\forall l \in \mathbb{N}.\ \texttt{goto}\ l \in \mathbb{I}_{\mathrm{MASM}}$ - jump to location l

- $\forall r \in \mathbb{B}^5, l \in \mathbb{N}.\ \texttt{ifnez}\ r\ \texttt{goto}\ l \in \mathbb{I}_{\text{MASM}}$ - conditional jump to location l
- $\mathbb{I}^{nocf}_{\text{MIPS}} \subset \mathbb{I}_{\text{MASM}}$ - all non-control-flow MIPS instructions

We call all MASM instructions that are not MIPS instructions *MASM macros*.

Programs

A MASM program π_μ is modeled as a partial mapping from procedure names to *procedure tables*. A procedure table contains information on the number of input parameters $npar$, the procedure $body$ in case it is not an external function (as denoted by the keyword `extern`) as well as the callee-save registers for that procedure encoded in a list of register indices named $uses$.

$$ProcT \stackrel{def}{\equiv} (npar : \mathbb{N}, body : \mathbb{I}^*_{\text{MASM}} \cup \{\texttt{extern}\}, uses : (\mathbb{B}^5)^*)$$

Thus we define the type of a MASM program $\pi_\mu \in Prog_{\text{MASM}}$ formally as follows.

$$Prog_{\text{MASM}} \stackrel{def}{\equiv} Names \rightharpoonup ProcT$$

There are certain validity conditions on MASM programs. First of all the MASM macros in the body of a procedure must be well-formed.

Definition 29 (MASM Macro Well-formedness) *For a MASM macro s at position j in the body of a procedure p of program π_μ we say that the satement is well-formed, if (i) parameter accesses must be within the right range, (ii) (conditional) jumps via `goto` may not leave the current procedure, (iii) only procedures declared in π_μ may be called, and (iv) every procedure ends with the `ret` macro, which must not appear elsewhere.*

$$\begin{aligned} wfm_{\text{MASM}}(s, j, p, \pi_\mu) \equiv \quad & \textit{(i)} \quad s \in \{\texttt{lparam}\ r\ i, \texttt{sparam}\ r\ i\} \implies i \leq \pi_\mu(p).npar \\ & \textit{(ii)} \quad s \in \{\texttt{goto}\ l, \texttt{ifnez}\ r\ \texttt{goto}\ l\} \implies l \leq |\pi_\mu(p).body| \\ & \textit{(iii)} \quad s = \texttt{call}\ P \implies \pi_\mu(P) \neq \bot \\ & \textit{(iv)} \quad s = \texttt{ret} \iff j = |\pi_\mu(p).body| \end{aligned}$$

Now we can give the complete definition of what it needs for a MASM program to be well-formed.

Definition 30 (MASM Program Well-formedness) *A MASM program π_μ is well-formed, if (i) all of its procedures contain at least one instruction, (ii) all contained MASM macros are well-formed, and (iii) contains a main procedure "`_start`" which sets up the stack. It takes no input parameters and has an empty $uses$ list.*

$wfprog_{\text{MASM}}(\pi_\mu) \equiv$

$$\begin{aligned} \forall p \in Names.\ \pi_\mu(p) \neq \bot \implies & \\ \textit{(i)} \quad & |\pi_\mu(p).body| > 0 \\ \textit{(ii)} \quad & \forall j \in [1 : |\pi_\mu(p).body|]. \\ & \qquad \pi_\mu(p).body[j] \notin \mathbb{I}^{nocf}_{\text{MIPS}} \implies wfm_{\text{MASM}}(\pi_\mu(p).body[j], j, p, \pi_\mu) \\ \textit{(iii)} \quad & \pi_\mu(\texttt{\_start}) \neq \bot \wedge \pi_\mu(\texttt{\_start}).npar = |\pi_\mu(\texttt{\_start}).uses| = 0 \end{aligned}$$

Configuration

We define the configuration of a MASM machine.

Definition 31 (MASM configuration) *The state of the MASM machine c_μ is a record*

$$c_\mu = (c, m) \in \mathbb{C}_{\text{MASM}}$$

with a memory component $m : \mathbb{B}^{32} \to \mathbb{B}^8$ and a MASM core $c \in \mathbb{K}_{\text{MASM}}$ that contains the following sub-components.

- $c.cpu : \mathbb{K}_{\text{MIPS}}$ *- the state of the MIPS processor core*
- $c.stack : Stack_{\text{MASM}}$ *- a stack abstraction (to be defined below)*
- $c.\pi : Prog_{\text{MASM}}$ *- a MASM program*
- $c.\texttt{SBA} : \mathbb{B}^{32}$ *- the stack base address, from where on the stack is stored in memory*

We use shorthands s_μ, π_μ, and $\texttt{SBA}_\mu$ for the respective components $c_\mu.c.stack$, $c_\mu.c.\pi$, and $c_\mu.c.\texttt{SBA}$. By $h_\mu \in \mathbb{H}_{\text{MIPS}}$ we denote a projection from MASM to ISA state which is defined as follows.

$$h_\mu.c = c_\mu.c.cpu \wedge h_\mu.m = c_\mu.m$$

Note that π_μ and $\texttt{SBA}_\mu$ are static components of the configuration, i.e., they do not change their values and can thus be seen as parameters for executing MASM computations. Also note that we altered the definition of MASM state compared to [Sha12]. In particular we embedded our MIPS ISA configuration in the MASM state and explicitely modeled parameters π_μ and $\texttt{SBA}_\mu$ as part of $\mathbb{C}_{\text{MASM}}$. Out of convenience h_μ still contains the program counter although it is never used in the MASM semantics.

Definition 32 (MASM Stack) *The stack of a MASM configuration consists only of a* no-frame *when the stack is not set up. Then we only save the current procedure's name p and the current location loc in its body.*

$$frame^{no}_{\text{MASM}} \equiv (p : Names, loc : \mathbb{N})$$

Otherwise the stack is modeled as a list of dynamic MASM stack frames.

$$frame_{\text{MASM}} \equiv (p : Names, loc : \mathbb{N}, pars : (\mathbb{B}^{32})^*, saved : (\mathbb{B}^{32})^*, lifo : (\mathbb{B}^{32})^*)$$

Besides the procedure name and current location of the program execution we store the following information.

- $pars$ *- contains a list of parameters passed to an instance of p.*
- $saved$ *- is a buffer to store the contents of callee-save registers. Note that in MASM semantics the callee-save registers defined in the $uses$ list of a procedure will be saved automatically on calling that procedure and restored when returning from that procedure.*

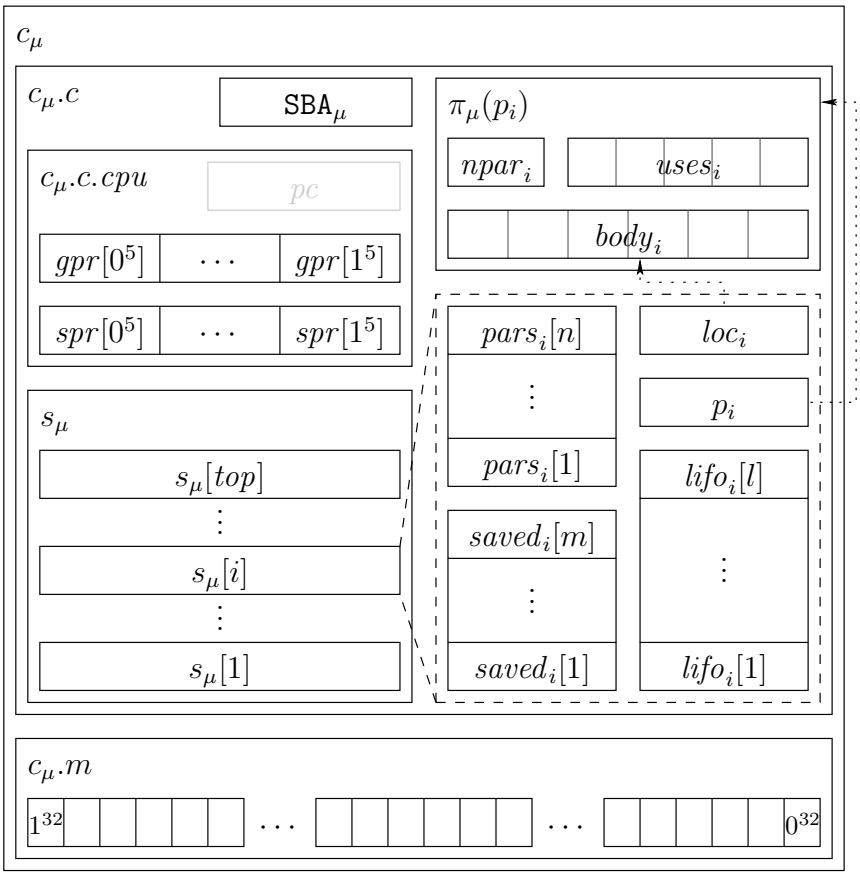

Figure 16: Illustration of the MASM configuration with a set up stack. The top of the *lifo* in frame i is indexed by 1. We have $l = |lifo_i|$, $m = |uses_i|$, and $n = npar_i$. The next statement to be executed in frame i is selected according to loc_i from entry p_i in the procedure table. CPU component pc is not used in MASM.

- *lifo - serves as the actual* stack *component which is used to store temporary data and procedure input parameters in a* last in, first out *manner.*

Thus the stack type is defined as follows.

$$Stack_{\mathrm{MASM}} \equiv frame^{no}_{\mathrm{MASM}} \cup frame^{*}_{\mathrm{MASM}}$$

See Fig. 16 for an illustration of the MASM confguration. Formally the MASM stack is set up if we have at least one proper MASM frame in the stack component, which we denote by the following predicate.

$$is\_set\_up(c_\mu) \stackrel{def}{\equiv} s_\mu \in frame^{*}_{\mathrm{MASM}} \wedge |s_\mu| > 0$$

For $/is\_set\_up(c_\mu)$ the stack consists only of a no-frame. In case it is set up, the top stack frame is the last element in s_μ. We abbreviate its index by $top \equiv |s_\mu|$. We speak of an

empty stack if it is set up with only one frame, that has an empty *lifo* as well as empty buffers for saved registers and input parameters.

$$is\_empty(c_\mu) \stackrel{def}{\equiv} is\_set\_up(c_\mu) \wedge top = 1 \wedge s_\mu[1].lifo = s_\mu[1].saved = s_\mu[1].pars = \varepsilon$$

We use the abbreviations given below if the stack is set up.

$$\begin{array}{lcllcl} p_i & \equiv & s_\mu[i].p & loc_i & \equiv & s_\mu[i].loc \\ pars_i & \equiv & s_\mu[i].pars & saved_i & \equiv & s_\mu[i].saved \\ lifo_i & \equiv & s_\mu[i].lifo & npar_i & \equiv & \pi_\mu[p_i].npar \\ body_i & \equiv & \pi_\mu[p_i].body & uses_i & \equiv & \pi_\mu[p_i].uses \end{array}$$

If the stack is not set up, p_1 and loc_1 denote the components $s_\mu.p$ and $s_\mu.loc$ and we have $top \equiv 1$. Using these shorthands we define the current instruction $I(c_\mu)$ to be executed in configuration c_μ.

$$I(c_\mu) \stackrel{def}{\equiv} body_{top}[loc_{top}]$$

Observe that this is short for $c_\mu.c.\pi[c_\mu.c.s[|c_\mu.c.s|].p].body[c_\mu.c.s[|c_\mu.c.s|].loc]$, so it seems we have found the right abbreviations to argue about MASM configurations in a succinct way.

Definition 33 (Well-formed MASM configuration) *A MASM configuration c_μ is well-formed if it (i) contains a well-formed MASM program, if (ii) the current procedure information and location counter are well-defined, and if (iii) for all stack frames exactly the specified number of parameters and callee-save registers are stored in the corresponding components.*

$$\begin{array}{lcll} wf_{\mathrm{MASM}}(c_\mu) & \equiv & (i) & wfprog_{\mathrm{MASM}}(\pi_\mu) \\ & & (ii) & \forall i \in [1 : top].\ \pi_\mu(p_i) \neq \bot \wedge loc_i \in [1 : |body_i|] \\ & & (iii) & is\_set\_up(c_\mu) \Longrightarrow \forall i \in [1 : top].\ |pars_i| = npar_i \wedge |saved_i| = |uses_i| \end{array}$$

Note that the last condition is necessary not only to have well-defined semantics, but also to be able to implement the abstract stack in memory where only a limited amount of space is reserved for storing the parameters and callee-save registers of a procedure. Theoretically the size of the *lifo* is not bounded, however in the section about simulation between MASM and ISA, we will bound the size of the *lifo*, and the stack respectively.

MIPS Assembler Calling Convention

The call procedure in MASM follows certain conventions that shall be described below. First of all certain general purpose registers of the MIPS processor core have special meaning. The details are depicted in Table 4.3. Note that this is a different setting than in [Sha12]. We changed the register association to adapt it to the MIPS *o32* calling convention [The06]. In particular the stack and base pointers are now stored in registers 29 and 30. The number of registers used for input parameters is limited to four and the return value of a procedure call is stored in register 2. We omitted the possibility of

Alias	⟨Index⟩	Usage
$zero$	0	always contains 0^{32}
t_1	1	temporary values
rv	2	procedure return value
t_2	3	temporary values
$i_1 \dots i_4$	$4, \dots, 7$	input arguments for procedure calls
$t_3 \dots t_{14}$	$8 \dots 15, 24 \dots 27$	temporary values
$sv_1 \dots sv_8$	$16 \dots 23$	callee-save registers
gp	28	global memory pointer
sp	29	stack pointer
bp	30	stack frame base pointer
ra	31	return address

Table 4.3: Intended usage of the 32 general purpose registers in MASM for MIPS

64-bit return values that would be stored in two registers. Also registers 26 and 27 do not have any special meaning in the framework of this thesis. However we introduced the pointer to the global memory region that is stored in register 28. We explicitely state which registers are the callee-save registers (16-23) that must be preserved across procedure calls by the programmer. Note that stack and base pointers as well as the return address are only relevant for the implementation of the MASM semantics in MIPS ISA, as MASM abstracts from the actual stack structure and from program-counter-based control-flow. Later we will have software conditions that forbid MASM programs accessing these registers.

Concerning the procedure call we have four calling conventions CC.1 to CC.4. They read as follows.

CC.1 In a procedure call up to four input parameters may be passed through the general purpose registers i_1 to i_4.

CC.2 Excess parameters must be passed via the caller's $lifo$. Parameters must be pushed on the stack in reverse order such that in the end the first excess parameter, i.e., the fifth parameter, resides on the top of $lifo$. In the implementation of the stack there is space reserved for the input parameters passed through the four registers. Thus the size of the memory region in the implementation devoted to storing the parameters of stack frame i is actually always equal to $npar_i$. All excess parameters that were pushed on the stack when there were more than four inputs to procedure p are consumed (popped) from the $lifo$ by a call of p and become part of the parameter space of the new stack frame.

CC.3 Before the return from a called procedure p all callee-save registers of p as well as the global memory pointer gp must be restored with the contents they had when p was entered. This must also be guaranteed for sp, bp, and ra by every MASM implementation. Instead of adding code for the saving and restoring of

callee-save registers to the body of p, the programmer may specify any modified callee-save register in the $uses$ list of p. Then the compiler takes care of saving and restoring these registers.

CC.4 The return value from a procedure call is passed through register rv.

Within a MASM program execution the programmer-independent parts of these conventions are obeyed by the Macro Assembler semantics and its implementation by construction. However, they become very important as soon as we mix MASM with other higher languages such as C and treat inter-language calls and returns. Later we will use a similar calling convention for the implementation of C-IL programs, thus allowing in principle for calling C-IL programs from MASM and vice versa. Indeed, Andrey Shadrin established such a mixed C-IL+MASM semantics [Sha12].

Transition Function

We execute MASM programs using the transition function:

$$\delta_{\text{MASM}} : \mathbb{C}_{\text{MASM}} \to (\mathbb{C}_{\text{MASM}} \cup \{\bot\})$$

It maps MASM configurations c_μ either to the next state c'_μ or to a fail-state $\bot$ in case a software condition was violated. These software conditions prevent the following situations that we consider run-time errors:

1. A stack instruction, i.e., `push`, `pop`, `lparam`, and `sparam`, or a procedure `call`, resp. return (`ret`), are executed while the stack is not set up.
2. The stack registers sp or bp are accessed while the stack is set up and not empty.
3. A `pop` instruction targets an empty $lifo$.
4. The return address register ra is accessed at any time.
5. The assembly code does produce an exception, i.e., an illegal instruction execution, misalignment, or overflow occurred.[4]

For the first three conditions we distinguish whether the stack is set up or not. In the latter case the following predicate detects run-time errors.

$$vio^{no}_{\text{MASM}}(c_\mu) \stackrel{def}{\equiv} /is\_set\_up(c_\mu) \wedge \exists P, r, i.\ I(c_\mu) \in \left\{ \begin{array}{l} \texttt{push}\ r, \texttt{pop}\ r, \texttt{lparam}\ r\ i, \\ \texttt{sparam}\ r\ i, \texttt{call}\ P, \texttt{ret} \end{array} \right\}$$

Popping from an empty $lifo$ is denoted as follows.

$$pop\_empty(c_\mu) \stackrel{def}{\equiv} I(c_\mu) = \texttt{pop}\ r \wedge lifo_{top} = \varepsilon$$

[4]Note that, as we excluded MIPS control flow instructions from $\mathbb{I}_{\text{MASM}}$ including sysc, there cannot be system call interrupts. Moreover the MASM semantics presented here omits treating the external event vector of the MIPS processor. This is treated in future work.

In general registers can be accesses by either MASM macro instructions or regular MIPS instructions. For some set $R \subseteq \mathbb{N}_{32}$ of register indices we can define the following predicates denoting an access to registers from R in the next step from state c_μ.

$$\begin{aligned} acc\_macro(c_\mu, R) &\stackrel{def}{\equiv} \exists r, i.\, I(c_\mu) \in \left\{ \begin{array}{l} \texttt{push}\, r, \texttt{pop}\, r, \\ \texttt{lparam}\, r\, i, \\ \texttt{sparam}\, r\, i \end{array} \right\} \wedge \langle r \rangle \in R \\ acc\_instr(c_\mu, R) &\stackrel{def}{\equiv} I(c_\mu) \in \mathbb{I}^{nocf}_{\mathrm{MIPS}} \wedge \{\langle rd(I(c_\mu))\rangle, \langle rs(I(c_\mu))\rangle, \langle rt(I(c_\mu))\rangle\} \cap R \neq \emptyset \\ acc(c_\mu, R) &\stackrel{def}{\equiv} acc\_macro(c_\mu, R) \vee acc\_instr(c_\mu, R) \end{aligned}$$

Then the predicate $vio^{st}_{\mathrm{MASM}}(c_\mu)$, denoting run-time errors 2 and 3, is defined by:

$$vio^{st}_{\mathrm{MASM}}(c_\mu) \stackrel{def}{\equiv} is\_set\_up(c_\mu) \wedge (pop\_empty(c_\mu) \vee /is\_empty(c_\mu) \wedge acc(c_\mu, \{sp, bp\}))$$

We define a JISR predicate for MASM in order to detect the forbidden exceptions described above using an empty external event vector and aligned program counter[5] 0^{32}.

$$jisr(c_\mu) \stackrel{def}{\equiv} I(c_\mu) \in \mathbb{I}^{nocf}_{\mathrm{MIPS}} \wedge jisr(h_\mu.c[pc := 0^{32}], I(c_\mu), 0^{256})$$

Finally we collect all run-time errors in the predicate

$$vio_{\mathrm{MASM}}(c_\mu) \stackrel{def}{\equiv} vio^{no}_{\mathrm{MASM}}(c_\mu) \vee vio^{st}_{\mathrm{MASM}}(c_\mu) \vee acc(c_\mu, \{ra\}) \vee jisr(c_\mu)$$

and we let δ evaluate to $\perp$ whenever such a violation occurs.

$$vio_{\mathrm{MASM}}(c_\mu) \Longleftrightarrow \delta_{\mathrm{MASM}}(c_\mu) = \perp$$

In the following we assume the absence of software condition violations and a valid program π_μ. Then the MASM instructions have the following effects.

- `push` r — adds the value from GPR r to the front of the *lifo*.
- `pop` r — removes the first value from the *lifo* and saves it in GPR r.
- `lparam` r i — reads the i-th parameter of the current procedure into GPR r
- `sparam` r i — takes the value from register r and saves it in place of the i-th parameter of the current procedure.
- `call` P — calls MASM procedure P.
- `ret` — returns from the current MASM procedure to the caller.

[5] Remember that we kept the program counter in the processor core for convenience. Since we are using the MIPS predicate *jisr* in the definition we need to make sure the undefined pc component does not produce a misaligment interrupt. The actual instruction to be executed is not fetched using the pc, but taken from the MASM program.

- `goto` l — jumps to location l in the current procedure.
- `ifnez` r `goto` l — jumps to location l if the value in GPR r is not equal to zero.

The effects of plain MIPS assembly instructions are unchanged and we rely on the definitions of our MIPS ISA model in case $I(c_\mu) \in \mathbb{I}^{nocf}_{\text{MIPS}}$, using the shorthand

$$h'_\mu \equiv \delta_{instr}(h_\mu, I(c_\mu))$$

to denote the next state of the processor configuration. Note that we are not using the program counter to determine the MIPS instruction to be executed next. Instead it is loaded from the current location in the program code. In the following we define the MASM semantics separately for each component. This complements the definition of [Sha12], where the semantics are specified with a focus on the different MASM operations. Let $c'_\mu = \delta_{\text{MASM}}(c_\mu)$ denote the next MASM configuration.

$$c'_\mu.m = \begin{cases} h'_\mu.m & : \quad I(c_\mu) \in \mathbb{I}^{nocf}_{\text{MIPS}} \\ c_\mu.m & : \quad \text{otherwise} \end{cases}$$

Abstracting from the stack region, the physical memory is only updated by plain assembly instructions. Also special purpose registers cannot be modified by macro steps.

$$c'_\mu.c.cpu.spr = \begin{cases} h'_\mu.c.spr & : \quad I(c_\mu) \in \mathbb{I}^{nocf}_{\text{MIPS}} \\ h_\mu.c.spr & : \quad \text{otherwise} \end{cases}$$

We do not define a new value for the program counter here, because we completely abstract from this control flow entity via the `goto` and `ifnez` macros and no MIPS control-flow instructions are not in $\mathbb{I}^{nocf}_{\text{MIPS}}$. For all $r \in \mathbb{B}^5$ we define the next state of the general purpose register. In case of plain assembly instructions we procede as above. Macro instructions that can modify GPRs are

1. `pop` r, which pops a value from the *lifo* into register r,
2. `lparam` r i, which reads parameter i of the current procedure and stores it in register r, and
3. `ret`, which returns from a procedure call, thus restoring all callee-save registers specified in the corresponding $uses$ list according to CC.3. Moreover we havoc all other registers besides $zero$ and rv with an unspecified boolean value U, thus preventing the programmer from relying on the content of any register that is not callee-save after a procedure call and return.
4. `call` P, which performs a procedure call. Similarly to the case of `ret` we havoc all registers on whose values the programmer of the callee function should not rely on after returning from the function call. This also serves to hide implementation details of the `call` routine since we do not disclose the values of the temporary registers after the procedure call was executed.

Let $CS \equiv \{sv_1, \ldots, sv_8, gp\}$ be the indices of the registers whose values must be restored by the MASM programmer before returning from a procedure. Moreover let function $npr(P) = \min\{4, \pi_\mu[P].npar\}$ compute the number of input parameters passed in registers to a given procedure P in the MASM program.

$$c'_\mu.c.cpu.gpr(r) = \begin{cases} h'_\mu.gpr(r) & : \quad I(c_\mu) \in \mathbb{I}^{nocf}_{\text{MIPS}} \\ lifo_{top}[1] & : \quad I(c_\mu) = \texttt{pop}\ r \\ pars_{top}[i] & : \quad I(c_\mu) = \texttt{lparam}\ r\ i \\ saved_{top}[j] & : \quad I(c_\mu) = \texttt{ret}\ \wedge \langle r \rangle \in CS \wedge uses_{top}[j] = r \\ U & : \quad U \in \mathbb{B}^{32} \wedge \\ & \quad \left(\begin{array}{l} I(c_\mu) = \texttt{ret}\ \wedge \langle r \rangle \notin CS \cup \{zero, rv\} \\ \vee\, \exists P.\, I(c_\mu) = \texttt{call}\ P \\ \quad \wedge \langle r \rangle \notin CS \cup \{zero, i_1, \ldots, i_{npr(P)}\} \end{array} \right) \\ h_\mu.gpr(r) & : \quad \text{otherwise} \end{cases}$$

The stack component remains to be treated. First of all we state when the number of stack frames is modified. Only by `call` and `ret` statements it can be incremented, or decremented respectively.

$$|c'_\mu.c.stack| = \begin{cases} |s_\mu| + 1 & : \quad \exists P.\, I(c_\mu) = \texttt{call}\ P \\ |s_\mu| - 1 & : \quad I(c_\mu) = \texttt{ret} \\ 1 & : \quad s_\mu \in frame^{no}_{\text{MASM}} \\ |s_\mu| & : \quad \text{otherwise} \end{cases}$$

A switch between no-frames and set-up stacks can only occur when one of the stack registers sp and bp is modified. Note that for violation-free programs this is only allowed when the stack is empty or not set up. This mechanism is useful for initialization of the stack by the `_start` procedure which has no inputs and an empty $uses$ list. The stack is then considered to be set up iff both stack and base pointer point to the base address of the first frame in memory which is located one word below the stack base address in case there are no input parameters (cf. Fig. 17). We overload the is_set_up predicate for MIPS configuration h, and a stack base address sba.

$$is\_set\_up(h, sba) \stackrel{def}{\equiv} h.gpr(sp) = h.gpr(bp) = sba -_{32} 4_{32}$$

Then the stack set up and destruction is modeled as follows.

$$c'_\mu.stack \in \begin{cases} frame_{\text{MASM}} & : \quad is\_set\_up(h'_\mu, \texttt{SBA}_\mu) \\ frame^{no}_{\text{MASM}} & : \quad /is\_set\_up(h'_\mu, \texttt{SBA}_\mu) \end{cases}$$

This definition covers also the cases where the stack type does not change. While the stack is set up and has type $frame^{*}_{\text{MASM}}$ registers sp and bp contain $\texttt{SBA}_\mu -_{32} 4_{32}$. Similarly, while these registers do not point to the base of the start frame the stack keeps type $frame^{no}_{\text{MASM}}$.

Now we define the five stack components for the next state. We use primed versions of the shorthands for procedure parameters and stack components in frame i when we refer to them wrt. configuration c'_μ. The definitions are made for all frames $i \in \mathbb{N}$ within the following ranges.

$$\begin{array}{rcl} s_\mu \in \mathit{frame}^{\mathit{no}}_{\mathrm{MASM}} & \Longrightarrow & i = 1 \\ \exists P.\, I(c_\mu) = \texttt{call}\ P & \Longrightarrow & i \in [1 : |s_\mu| + 1] \\ I(c_\mu) = \texttt{ret} & \Longrightarrow & i \in [1 : |s_\mu| - 1] \\ \text{otherwise} & \Longrightarrow & i \in [1 : |s_\mu|] \end{array}$$

We define the location and procedure in frame i of c'_μ as follows.

$$\mathit{loc}'_i = \begin{cases} 1 & : \quad \exists P.\, I(c_\mu) = \texttt{call}\ P \wedge i = \mathit{top} + 1 \\ l & : \quad i = \mathit{top} \wedge (\, I(c_\mu) = \texttt{goto}\ l \\ & \qquad \vee \exists r.\, I(c_\mu) = \texttt{ifnez}\ r\ \texttt{goto}\ l \wedge h_\mu.\mathit{gpr}(r) \neq 0^{32}\,) \\ \mathit{loc}_i & : \quad i < \mathit{top} \\ \mathit{loc}_i + 1 & : \quad \text{otherwise} \end{cases}$$

The location is only modified in the top frame or set to one in new frames that are created at a procedure calls. If no control-flow macro is executed the location is simply incremented.

$$p'_i = \begin{cases} P & : \quad I(c_\mu) = \texttt{call}\ P \wedge i = \mathit{top} + 1 \\ p_i & : \quad \text{otherwise} \end{cases}$$

We only set the procedure of a new stack frame created during a procedure call.

The next state for components *pars*, *saved* and *lifo* is only defined if $\mathit{is\_set\_up}(c'_\mu)$ holds, i.e., if the stack is set up or not being destroyed by the step. Let $\mathit{npar}'_i = \pi_\mu(p'_i).\mathit{npar}$ in:

$$\mathit{pars}'_i[j] = \begin{cases} h_\mu.\mathit{gpr}(r) & : \quad I(c_\mu) = \texttt{sparam}\ r\ j \wedge i = \mathit{top} \\ \mathit{lifo}_{\mathit{top}}[j-4] & : \quad \exists P.\, I(c_\mu) = \texttt{call}\ P \\ & \qquad \wedge\ j \in (4 : \mathit{npar}'_i] \wedge i = \mathit{top} + 1 \\ U & : \quad U \in \mathbb{B}^{32} \wedge j \in \mathbb{N}_{\mathit{npar}'_i} \wedge \\ & \qquad \left(\begin{array}{l} \exists P.\, I(c_\mu) = \texttt{call}\ P \wedge i = \mathit{top} + 1 \\ \vee\, / \mathit{is\_set\_up}(c_\mu) \wedge \mathit{is\_set\_up}(c'_\mu) \wedge i = 1 \end{array} \right) \\ \mathit{pars}_i[j] & : \quad \text{otherwise} \end{cases}$$

The parameters component can be updated with a register value via `sparam` to store away an input parameter to the stack. Moreover it is written upon a procedure call to pass excess parameters for which there are no more registers available and to reserve space for the parameters passed through registers. Its values are also unspecified when the stack is being set up. Note that compared to [Sha12] we changed the order in which values are saved in the *lifo*. The head is now $\mathit{lifo}[1]$ which makes the formulation of

parameter passing easier. Using $uses'_i = \pi_\mu(p'_i).uses$ we define the next state for the callee-save registers component.

$$saved'_i[j] = \begin{cases} h_\mu.gpr(uses'_i[j]) & : \quad \exists P.\ I(c_\mu) = \texttt{call}\ P \wedge i = top + 1 \wedge j \leq |uses'_i| \\ U & : \quad /is\_set\_up(c_\mu) \wedge is\_set\_up(c'_\mu) \wedge U \in \mathbb{B}^{32} \wedge i = 1 \\ saved_i[j] & : \quad \text{otherwise} \end{cases}$$

The components to store callee-save registers are only modified in the newly created frame of a procedure call. We save the register values according to the corresponding $uses$ list. Upon stack setup, the content of $saved$ is unspecified. The *lifo* is manipulated in the obvious way by `push` and `pop` operations. When the stack is set up and in a newly called frame it is empty. When we call a procedure all parameters passed through *lifo* are popped.

$$lifo'_i = \begin{cases} h_\mu.gpr(r) \circ lifo_i & : \quad \exists r.\ I(c_\mu) = \texttt{push}\ r \wedge i = top \\ tl(lifo_i) & : \quad \exists r.\ I(c_\mu) = \texttt{pop}\ r \wedge i = top \\ \varepsilon & : \quad /is\_set\_up(c_\mu) \wedge is\_set\_up(c'_\mu) \wedge i = 1 \\ & \quad \vee\ \exists P.\ I(c_\mu) = \texttt{call}\ P \wedge i = top + 1 \\ pop(lifo_i, npar'_{i+1} - 4) & : \quad \exists P.\ I(c_\mu) = \texttt{call}\ P \wedge npar'_{i+1} > 4 \wedge i = top \\ lifo_i & : \quad \text{otherwise} \end{cases}$$

This completes the definition of MASM semantics.

4.2.2 MASM Assembler Consistency

We do not present an implementation of MASM in the scope of this thesis but just assume a function

$$asm : \mathbb{I}_{\text{MASM}} \to \mathbb{I}^*_{\text{MIPS}}$$

which represents the code generation function of the assembler. For a given MASM statement s it returns a sequence of MIPS instructions that is supposed to implement s. A sketch of a realisation of asm for VAMP ISA can be found in [Sha12]. Although we do not specify a MIPS assembler here, we nevertheless want to reproduce the assembler consistency relation for our MIPS-based MASM semantics using the same stack layout as in [Sha12] which we define below.

Stack Layout

The implementation of the stack in memory is depicted in Fig. 17. There we can identify the following structure of a frame. Note that the stack is growing from top to bottom.

- A MASM frame i in memory is identified by a base address $base(i)_{32}$.

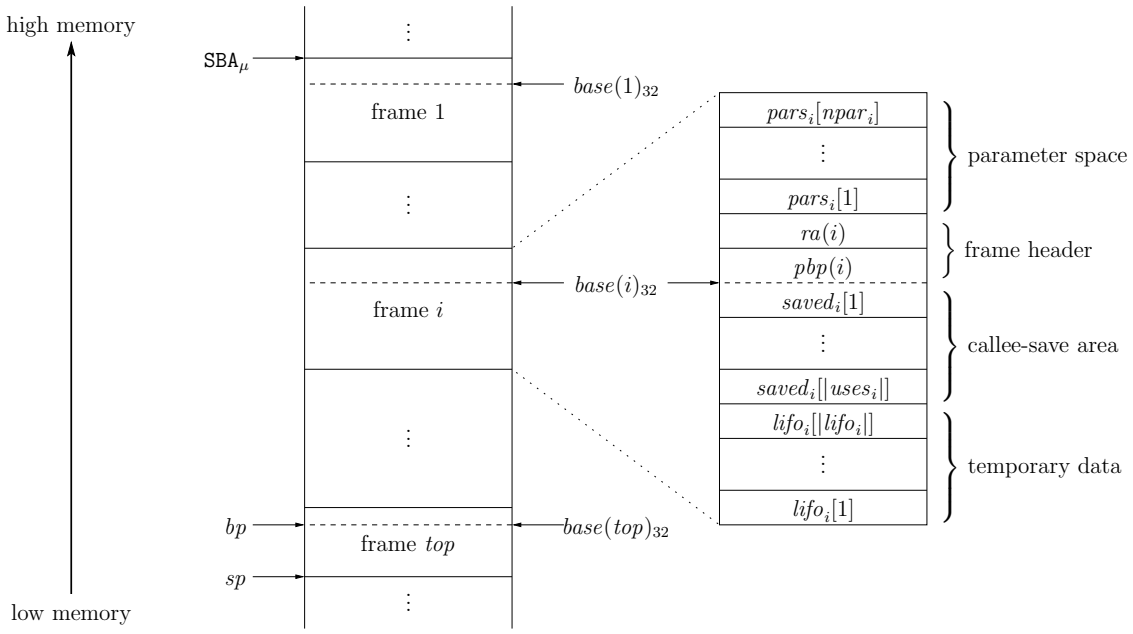

Figure 17: Illustration of the MASM stack layout in memory. The base address of each frame is shown as a dashed line. Note that for some procedures the areas reserved for parameters, callee-save registers and the *lifo* may have size zero.

- Below the base address we store the callee save registers. We also have space reserved for storing callee-save registers on the lowest frame $i = 1$, which was not the case in [Sha12].
- Below the $saved_i$ data the $lifo_i$ stack for temporary data and parameters is located, growing also towards the bottom of the stack area.
- Parameters $j > 4$ for a call of p_i are pushed on the *lifo* of the caller in reverse order. Upon the call they become part of stack frame i, thus they are located above the base address $base(i)$. We also reserve room for up to four parameters being passed through registers.
- Below the parameter space resides the frame header. It contains first the return address, which specifies where instruction execution should continue when the return statement of p_i is invoked. The field of the header below the return address is allocated at the base address $base(i)$ of the frame. It contains a link to the previous frame base and is thus called *previous base pointer* (pbp).
- Finally we keep the top stack frame base in the *base pointer* register bp. The *stack pointer* register sp identifies the top element of $lifo_{top}$.
- All addresses of stack components are word-aligned.

The space between base addresses of frames i and $i+1$ contains the saved registers and the *lifo* of frame i as well as the parameters and frame header for $i+1$. We compute this distance for $i \in [1 : top)$ as follows.

$$dist(i) \equiv |uses_i| + |lifo_i| + npar_{i+1} + 2$$

Then the base address of frame i is easily defined recursively. In the base case we have an offset from the stack base address for the parameters and return address passed to the first function frame. Note that for convenient notation we model the base addresses as natural numbers first. Later on a conversion of numbers x to bit strings x_{32} will be unavoidable.

$$base(i) \equiv \begin{cases} \langle \mathtt{SBA}_\mu \rangle - 4 \cdot npar_1 - 4 & : \quad i = 1 \\ base(i-1) - 4 \cdot dist(i-1) & : \quad i \in (1 : top] \end{cases}$$

The return address and previous base pointer of frame $i \in [1 : top]$ wrt. some ISA configuration h that implements the MASM program are denoted by:

$$ra(i) \equiv h.m_4((base(i)+4)_{32}) \qquad pbp(i) \equiv h.m_4(base(i)_{32})$$

Assembler Information

To define the assembler consistency relation and a simulation theorem for MASM we need to know more about the assembling process of a program π_μ than only the stack layout. In [Sha12] for this purpose an assembler information structure $info_\mu \in InfoT_{\mathrm{MASM}}$ was introduced. We resemble its definition with slight technical modifications below.

- $info_\mu.code \in \mathbb{I}^*_{\mathrm{MIPS}}$ — A list of MIPS instructions representing the assembled program.
- $info_\mu.off : Names \times \mathbb{N} \to \mathbb{N}_0$ — A function calculating the offset in the compiled code of the first instruction which implements a statement at the specified location in the given procedure. Note that offset 0 refers to instruction $info_\mu.code[1]$.
- $info_\mu.cba \in \mathbb{B}^{32}$ — the start address of the code region in memory
- $info_\mu.mss \in \mathbb{N}$ — the maximal stack size as a number of words

The first two components are depending on the assembler while the latter two must be specified by the programmer. We treat them as assembling information anyway.

Simulation Relation

We want to relate a MASM configuration c_μ to an ISA state h that implements the program π_μ using $info_\mu$. Formally we thus define a simulation relation

$$consis_{\mathrm{MASM}}(c_\mu, info_\mu, h)$$

stating the consistency between these entities. It is split in two sub-relations covering control and data consistency. The first part talks about control-flow and is thus concerned with the program counter and the return address. Let the following function compute the start address of the compiled code for the MASM statement at position loc in procedure p.

$$adr(info_\mu, p, loc) \stackrel{def}{\equiv} info_\mu.cba +_{32} (4 \cdot info_\mu.off(p, loc))_{32}$$

Definition 34 (MASM Control Consistency) *We define MASM consistency sub-relation $consis^{control}_{\text{MASM}}$, which treats the control-flow aspects of coupling a MASM configuration c_μ with an implementing MIPS configuration h taking into account compiler information $info_\mu$. It states that (i) the program counter of the MIPS machine must point to the start of the compiled code for the current statement in the MASM machine. In addition (ii) the return address of any stack frame is pointing to the compiled code for the next statement in the previous frame.*

$$\begin{aligned} consis^{control}_{\text{MASM}}(c_\mu, info_\mu, h) \equiv \quad & (i) \quad h.pc = adr(info_\mu, p_{top}, loc_{top}) \\ & (ii) \quad is\_set\_up(c_\mu) \implies \\ & \qquad \forall i \in (1, top].\ ra(i) = adr(info_\mu, p_{i-1}, loc_{i-1}) \end{aligned}$$

Data consistency is split into several parts covering registers, the memory, the code region as well as the stack. Register consistency is straight forward except for the stack and base pointers. For them the base pointer points to the base address of the top frame while the stack pointer points to the top element of the lifo in the top frame in case the stack is set up. Let $stktop = (base(top) - 4 \cdot (|uses_{top}| + |lifo_{top}|))_{32}$ in:

$$consis^{bp}_{\text{MASM}}(c_\mu, h) \stackrel{def}{\equiv} h.gpr(bp) = \begin{cases} base(top)_{32} & : \quad is\_set\_up(c_\mu) \\ h_\mu.gpr(bp) & : \quad \text{otherwise} \end{cases}$$

$$consis^{sp}_{\text{MASM}}(c_\mu, h) \stackrel{def}{\equiv} h.gpr(sp) = \begin{cases} stktop & : \quad is\_set\_up(c_\mu) \\ h_\mu.gpr(sp) & : \quad \text{otherwise} \end{cases}$$

Remember here that h_μ is the projection of c_μ to an ISA configuration. Note also that above we do not specify the values of stack and base pointer after returning from the lowest stack frame. Calling convention CC.3 however demands that they have the same values as when the MASM program was started. Therefore this issue becomes an additional verification condition on the compiler, when considering a mixed programming language semantics where different programs may call each other [Sha12].

Besides stack pointer, base pointer and return address register ra, all other registers always we have the same contents in ISA and MASM configuration. The return address register is used only by the implementation on the MIPS layer and may not be accessed by the MASM programmer, hence we do not need to specify a coupling relation for it.

$$\begin{aligned} & consis^{regs}_{\text{MASM}}(c_\mu, h) \stackrel{def}{\equiv} \\ & \quad \forall r \in \mathbb{B}^5.\ (r \notin \{sp, bp, ra\} \implies h.gpr(r) = h_\mu.gpr(r)) \wedge h.spr(r) = h_\mu.spr(r) \end{aligned}$$

Definition 35 (MASM Code Consistency) *For code consistency we require that (i) the assembled code in the assembler information is actually corresponding to the MASM program and that (ii) the assembled code is converted to binary format and resides in a contiguous region in the memory of the MIPS machine starting at the code base address. Let*

$$ad_j = (\langle info_\mu.cba \rangle + 4 \cdot j)_{32}$$

be the memory address of instruction j of the compiled program, then we define $consis^{code}_{\mathrm{MASM}}$ by:

$$\begin{aligned}
&consis^{code}_{\mathrm{MASM}}(c_\mu, info_\mu, h) \stackrel{def}{\equiv} \\
&\quad \textit{(i)} \quad \forall p \in Names, l.\ \pi_\mu(p) \neq \bot \wedge l \in [1 : |\pi_\mu(p).body|] \\
&\qquad\qquad \implies info_\mu.code[info_\mu.off(p,l)+1] = asm(\pi_\mu(p).body[l]) \\
&\quad \textit{(ii)} \quad \forall j \in [0 : |info_\mu.code|).\ info_\mu.code[j+1] = decode(h.m_4(ad_j))
\end{aligned}$$

Observe that the latter property forbids self-modifying MASM programs in case $info_\mu$ is fixed for a simulation between the MIPS and MASM machines.

Definition 36 (MASM Stack Consistency) *We have consistent stacks, if for every stack frame (i) the lifo, (ii) the saved registers, and (iii) the parameters are correctly embedded into the stack region according to the stack layout. Moreover (iv) in every frame except the lowest one the previous base pointer field contains the address of the base of the frame above.*

$$\begin{aligned}
&consis^{stack}_{\mathrm{MASM}}(c_\mu, h) \ \equiv\ \forall i \leq top. \\
&\quad \textit{(i)} \quad \forall j \in \mathbb{N}_{|lifo_i|}.\ h.m_4((base(i) - 4 \cdot (|uses_i| + |lifo_i| + 1) + 4j)_{32}) = lifo_i[j] \\
&\quad \textit{(ii)} \quad \forall j \in \mathbb{N}_{|uses_i|}.\ h.m_4((base(i) - 4j)_{32}) = saved_i[j] \\
&\quad \textit{(iii)} \quad \forall j \in \mathbb{N}_{npar_i}.\ h.m_4((base(i) + 4 + 4j)_{32}) = pars_i[j] \\
&\quad \textit{(iv)} \quad i > 1 \implies pbp(i) = base(i-1)_{32}
\end{aligned}$$

We use the shorthands CR and StR to represent the code region, or the region where the stack is allocated respectively. Let $msp_\mu \equiv \langle \mathtt{SBA}_\mu \rangle - info_\mu.mss + 1$ be the minimal integer value of the stack pointer (which should be non-negative for meaningful applications). Then we define:

$$CR \equiv [\langle info_\mu.cba \rangle : \langle info_\mu.cba \rangle + 4 \cdot |info_\mu.code|) \qquad StR \equiv [msp_\mu : \langle \mathtt{SBA}_\mu \rangle]$$

In the MASM semantics we abstract from the assembled code by π_μ and from the stack region by s_μ. However when the stack is not set up we allow the stack region to be accessed in the area of the first stack frame for the `_start` procedure. This area SF (start frame) contains only the frame header, since `_start` does not take input parameters and has an empty $uses$ list.

$$SF \ \equiv\ [\langle \mathtt{SBA}_\mu \rangle - 4 : \langle \mathtt{SBA}_\mu \rangle]$$

Now we demand memory consistency for all addresses but the code region and the remaining part of the stack region. This means conversely that the contents of these regions in the MASM semantics are meaningless.

$$consis^{mem}_{\mathrm{MASM}}(c_\mu, info_\mu, h) \stackrel{def}{\equiv} \forall ad \in \mathbb{B}^{32}.\ \langle ad \rangle \notin CR \cup (StR \setminus SF) \Longrightarrow h.m(ad) = h_\mu.m(ad)$$

Definition 37 (MASM Data Consistency) *The data consistency relation comprises the consistency between MIPS and MASM machine wrt. (i) stack pointer, (ii) base pointer, (iii) all other registers besides ra, (iv) the code region, (v) the stack, and (vi) the memory.*

$$consis^{data}_{\mathrm{MASM}}(c_\mu, info_\mu, h) \equiv \begin{array}{llll} (i) & consis^{bp}_{\mathrm{MASM}}(c_\mu, h) & (iv) & consis^{code}_{\mathrm{MASM}}(c_\mu, info_\mu, h) \\ (ii) & consis^{sp}_{\mathrm{MASM}}(c_\mu, h) & (v) & consis^{stack}_{\mathrm{MASM}}(c_\mu, h) \\ (iii) & consis^{regs}_{\mathrm{MASM}}(c_\mu, h) & (vi) & consis^{mem}_{\mathrm{MASM}}(c_\mu, info_\mu, h) \end{array}$$

Finally we define the overall simulation relation between MASM and MIPS as the conjunction of control and data consistency.

$$consis_{\mathrm{MASM}}(c_\mu, info_\mu, h) \stackrel{def}{\equiv} consis^{control}_{\mathrm{MASM}}(c_\mu, info_\mu, h) \wedge consis^{data}_{\mathrm{MASM}}(c_\mu, info_\mu, h)$$

Simulation Theorem

As we pointed out above the stack and code region of memory is not modeled in the MASM configuration as we abstract it to s_μ, and π_μ respectively. Thus any memory access to these areas could potentially break the MASM simulation. Moreover if the stack grows too big to fit into the stack region (stack overflow) the simulation breaks as well. As these issues are related to the simulation by the MIPS implementation and thus cannot be detected without knowing $info_\mu$, they are not covered by $vio_{\mathrm{MASM}}(c_\mu)$.

A stack overflow occurs if the top element of $lifo_{top}$ is stored at a location below the address specified by the minimal stack pointer value, i.e., outside of the stack region. Remember here that the stack is growing downwards.

$$stackovf(c_\mu, info_\mu) \stackrel{def}{\equiv} base(top) - 4 \cdot (|uses_{top}| + |lifo_{top}|) < msp_\mu$$

Self-modification or a direct stack accesses outside of the start frame are considered bad memory accesses and the start frame may only be accessed when the stack is not set up. Generally, memory can only be accessed by MIPS ISA memory instructions, since the macros work only on the stack abstraction and registers. With $ea \equiv ea(h_\mu.c, I(c_\mu))$, let

$$W(c_\mu) \stackrel{def}{\equiv} \begin{cases} \{ea, \ldots, ea +_{32} 3_{32}\} & : \quad \mathrm{sw}(I(c_\mu)) \vee swap(h_\mu, I(c_\mu)) \\ \emptyset & : \quad \text{otherwise} \end{cases}$$

be the addresses written by such a MIPS instruction in the MASM program. The addresses read by MASM statements can be obtained using the MIPS reads-set function R and we define a predicate to detect bad memory accesses.

$$\begin{aligned} badmemop(c_\mu, info_\mu) \stackrel{def}{\equiv}\ & I(c_\mu) \in \mathbb{I}^{nocf}_{\mathrm{MIPS}} \wedge mem(I(c_\mu)) \wedge \\ & \exists a \in R(h_\mu.c, I(c_\mu)) \cup W(c_\mu).\ \langle a \rangle \in CR \cup StR \setminus SF \vee is\_set\_up(c_\mu) \wedge \langle a \rangle \in SF \end{aligned}$$

We sum up the MASM software conditions below.

Definition 38 (MASM Software Conditions) *A MASM program execution can only be simulated by a MIPS ISA implementation if all reachable configurations obey the software conditions denoted by the following predicate. Given a MASM configuration c_μ and assembler information $info_\mu$, then (i) the next step may not produce a run-time error, (ii) the stack may not overflow nor be configured to wrap around at address 0^{32}, and (iii) the next step must not access the stack or code region directly.*

$$\begin{aligned} sc_{\mathrm{MASM}}(c_\mu, info_\mu) \equiv\ & \textit{(i)} && \delta_{\mathrm{MASM}}(c_\mu) \neq \bot \\ & \textit{(ii)} && /stackovf(c_\mu, info_\mu) \wedge msp_\mu \geq 0 \\ & \textit{(iii)} && /badmemop(c_\mu, info_\mu) \end{aligned}$$

Complementing the MASM software conditions we also define conditions for the MASM program execution on the MIPS level requiring that the assembled code is well-behaved. In particular here this means that it should not cause any interrupts. This can however only be achieved for *suitable* schedules of the MIPS machine where no non-maskable external interrupts (i.e., $reset$) do occur. For a given MIPS external event vector $eev \in \mathbb{B}^{256}$ we define predicate

$$suit^{\mathrm{MASM}}_{\mathrm{MIPS}}(eev) \stackrel{def}{\equiv} eev[0] = 0$$

which should hold for external event vectors of an implementing MIPS schedule. The absence of maskable external interrupts can be guaranteed by disabling all external interrupts for p. We define a well-formedness predicate for a MIPS configuration that allows MASM simulation without being interrupted by devices.

$$wf^{\mathrm{MASM}}_{\mathrm{MIPS}}(h) \stackrel{def}{\equiv} h.c.spr(sr)[\mathrm{dev}] = 0$$

Moreover in the code generation, especially for the macros, one has to take care that no misaligned or illegal code is produced. In order to maintain the aforementioned code invariant, we also demand that instructions are only fetched frmo the code region. Then we can demand that a MIPS computation step is well-behaved for MASM simulation by the following predicate.

$$wb^{\mathrm{MASM}}_{\mathrm{MIPS}}(h, eev) \stackrel{def}{\equiv} /jisr(h.c, I(h), eev) \wedge [\langle h.c.pc \rangle : \langle h.c.pc \rangle + 3] \subseteq CR$$

Note, that $wb^{\mathrm{MASM}}_{\mathrm{MIPS}}(h, eev)$ implies $suit^{\mathrm{MASM}}_{\mathrm{MIPS}}(eev)$. In our generalized simulation theory to be introduced in Sect. 5.2, the concepts of good behaviour of an implementation and suitability of inputs need not be connected. Suitability here is a necessary condition on the scheduling of external reset signals such that a simulation of MASM is possible in the first place. On the other hand, good behaviour of the implementation is something that the MASM simulation theorem guarantees for suitable schedules, namely that the computation is not interrupted at all and we only fetch from the code region.

We now can state our sequential simulation theorem between MASM and MIPS. In fact we only specify the simulation of one MASM step, which can be used for the induction step in a simulation between MIPS and MASM computations.

Theorem 2 (Sequential MASM Simulation Theorem) *Given a MASM starting configuration $c_{\mu 0} \in \mathbb{C}_{\mathrm{MASM}}$ that is (i) well-formed, a MIPS configuration $h_0 \in \mathbb{H}_{\mathrm{MIPS}}$ that is (ii) well-formed for MASM simulation and (iii) consistent to $c_{\mu 0}$ wrt. some $info_\mu \in InfoT_{\mathrm{MASM}}$.If (iv) the next step from $c_{\mu 0}$ fulfills the MASM software conditions and leads into a well-formed MASM state,*

$$\begin{array}{lll} \forall c_{\mu 0}, h_0, info_\mu\,. & \textit{(i)} & wf_{\mathrm{MASM}}(c_{\mu 0}) \\ & \textit{(ii)} & wf^{\mathrm{MASM}}_{\mathrm{MIPS}}(h_0) \\ & \textit{(iii)} & consis_{\mathrm{MASM}}(c_{\mu 0}, info_\mu, h_0) \\ & \textit{(iv)} & sc_{\mathrm{MASM}}(c_{\mu 0}, info_\mu) \wedge wf_{\mathrm{MASM}}(\delta_{\mathrm{MASM}}(c_{\mu 0})) \end{array}$$

then there exists an ISA computation that (i) starts in h_0 and leads into a well-formed state that is (ii) consistent with the MASM state obtained by stepping c_μ once. Moreover (iii) the implementing ISA computation is well-behaved and uses only suitable external event vectors where reset is off.

$$\begin{array}{lll} \Longrightarrow \quad \exists h \in \mathbb{H}^*, n \in \mathbb{N}_0, eev \in (\mathbb{B}^{256})^*. & & \\ & \textit{(i)} & h_1 = h_0 \wedge h_1 \longrightarrow^n_{\delta_{\mathrm{MIPS}}, eev} h_{n+1} \wedge wf^{\mathrm{MASM}}_{\mathrm{MIPS}}(h_{n+1}) \\ & \textit{(ii)} & consis_{\mathrm{MASM}}(\delta_{\mathrm{MASM}}(c_{\mu 0}), info_\mu, h_{n+1}) \\ & \textit{(iii)} & \forall i \in \mathbb{N}_n.\ wb^{\mathrm{MASM}}_{\mathrm{MIPS}}(h_i, eev_i) \wedge suit^{\mathrm{MASM}}_{\mathrm{MIPS}}(eev_i) \end{array}$$

Given the code generation function of a MASM assemler, we can prove this theorem by a case split on the next MASM statement, arguing about the correctness of the generated code. The theorem allows us to map any uninterrupted sequential MIPS ISA computation which is running out of a consistent configuration to a corresponding MASM computation, because due to determinism there is only one possible computation that the MIPS ISA can perform, namely the one which is simulating the MASM program execution.

Hypothesis *(iv)* demands that the MASM configuration resulting from the simulated step is well-formed. The well-formedness of a MASM configuration depends only on the well-formedness of the fixed program π_μ and on the size of the parameter and callee-save components in each stack frame. Hence it should be an easy lemma to show that the MASM semantics preserves well-formedness, however we do not present a proof in this thesis.

4.2.3 Concurrent MASM

Finally we want to instantiate our *Cosmos* model with the MASM semantics, establishing a concurrent system of n MASM computation units. Therefore we define a *Cosmos* machine $S^n_{\mathrm{MASM}} \in \mathbb{S}$ below. Later on (cf. Section 5.6.1) we want to prove a simulation between S^n_{MASM} and S^n_{MIPS} using the sequential simulation defined above in a concurrent setting. Moreover while all MASM units share the same program, they are running on different stacks with different stack base addresses but the same length. For the instantiation we therefore need to know $\pi \in \mathit{Prog}_{\mathrm{MASM}}$, $\mathit{info}_\mu \in \mathit{InfoT}_{\mathrm{MASM}}$, and $\mathtt{SBA}_r \in \mathbb{B}^{32}$ for all units $r \in \mathbb{N}_n$. For easy composition of the sequential simulation theorems we demand that the stacks must not overlap. We (re)define functions

$$\begin{aligned} CR &\equiv [\langle \mathit{info}_\mu.cba\rangle : \langle \mathit{info}.cba\rangle + 4 \cdot |\mathit{info}_\mu.code|) \\ \mathit{StR}_r &\equiv (\langle \mathtt{SBA}_r\rangle - \mathit{info}_\mu.mss : \langle \mathtt{SBA}_r\rangle] \\ \mathit{SF}_r &\equiv [\langle \mathtt{SBA}_r\rangle - 4 : \langle \mathtt{SBA}_r\rangle] \end{aligned}$$

to denote the individual regions for stack and start frame. Then the disjointness of stack regions is easily defined by:

$$\forall q, r \in \mathbb{N}_n.\ q \neq r \implies \mathit{StR}_r \cap \mathit{StR}_q = \emptyset$$

We now define the components of S^n_{MASM} one by one.

- $S^n_{\mathrm{MASM}}.\mathcal{A} = \{a \in \mathbb{B}^{32} \mid \langle a\rangle \notin CR \cup \bigcup_{u=1}^{n} \mathit{StR}_u \setminus \mathit{SF}_u\}$ and $S^n_{\mathrm{MASM}}.\mathcal{V} = \mathbb{B}^8$ — we have the same memory for the MASM system as the MIPS system, except that we cut out the forbidden address ranges for the MASM stack and code. Thus the condition $/\mathit{badmemop}$ demanded for MASM simulation is mostly covered by the ownership policy since it requires that accessed addresses must lie in $\mathcal{A}$. Only for the SF_u it must be checked that it is not accessed while the stack is set up.

- $S^n_{\mathrm{MASM}}.\mathcal{R} = \emptyset$ — since we cut out the code region from memory we do not need to make it read-only as in the MIPS instantiation. One could however think of global constants that are shared between the computation units. In such a case the constants' memory locations could be included here to prevent them via the ownership policy from write accesses.

- $S^n_{\mathrm{MASM}}.nu = n$ — We have n MASM computation units.

- $S^n_{\mathrm{MASM}}.\mathcal{U} = \mathbb{K}_{\mathrm{MASM}} \cup \{\bot\}$ — each computation unit is a MASM core, containing the CPU registers, the stack, program, and stack base address. We also include a fail-state $\bot$ to record run-time errors. Every unit r must be initially instantiated with program π and stack base address $\mathtt{SBA}_r$. If $C_0 \in \mathbb{C}_{S^n_{\mathrm{MASM}}}$ is the initial configuration of a MASM *Cosmos* machine then we have:

$$\forall r \in \mathbb{N}_n.\ C_0.u_r.\pi = \pi \wedge C_0.u_r.\mathtt{SBA} = \mathtt{SBA}_r$$

- $S^n_{\mathrm{MASM}}.\mathcal{E} = \{\varepsilon\}$ - in MASM we do not have any external inputs, however because many functions in the *Cosmos* model are define for some input, we provide a dummy one.

- $S^n_{\mathrm{MASM}}.reads$ — as we abstract from instruction fetch and stack memory accesses we only read memory via regular MIPS memory instructions. Thus we can reuse notation from the MIPS instantiation.

  $$S^n_{\mathrm{MASM}}.reads(u, m, \varepsilon) = R(c.cpu, I(u, m))$$

  Note above that a pair $(u, m) \in S^n_{\mathrm{MASM}}.\mathcal{U} \times (S^n_{\mathrm{MASM}}.\mathcal{A} \to S^n_{\mathrm{MASM}}.\mathcal{V})$ has exactly type $\mathbb{C}_{\mathrm{MASM}}$. Thus we can use it instead of a MASM configuration c_μ.

- $S^n_{\mathrm{MASM}}.\delta$ — naturally we employ MASM transition function δ_{MASM} in the instantiation, however, like in the MIPS case, we need to fill the partial memory that is given to the $S^n_{\mathrm{MASM}}.\delta$ as an input with dummy values, so that we can apply δ_{MASM} on it. Then with $(u', m') = \delta_{\mathrm{MASM}}(u, \lceil m \rceil)$ we can define the transition function for MASM computation units which returns the same new core configuration u' and the updated part of memory $m'|_{W(u,\lceil m \rceil)}$.

  $$S^n_{\mathrm{MASM}}.\delta(u, m, \varepsilon) = (u', m'|_{W(u,\lceil m \rceil)})$$

  If δ_{MASM} returns $\perp$ or $u = \perp$, then also $S^n_{\mathrm{MASM}}.\delta(u, m, \varepsilon) = \perp$.

- $S^n_{\mathrm{MASM}}.\mathcal{IO}$ — for concurrent MASM we assume a scenario where there are two ways to access shared memory. One way is to use the Compare-and-Swap instruction of the MIPS ISA. Moreover there is a set of 32-bit global variables that are shared between the MASM computation units. As also read and write instructions are atomic on the MASM level these variables can be safely accessed by regular MIPS read and write instructions without the need to acquire ownership of them first. Thus they are similar to the *volatile* variables of C programs.

  However, in order to avoid accidental shared variable accesses which are not expected by the programmer, we make the convention that the shared global variables must be accessed using the global variables pointer which is stored in register gp. The shared global variables are then identified by a set of word offsets $\mathit{off}_{gs} \subsetneq \mathbb{N}$ to the global variables pointer. For some offset $o \in \mathit{off}_{gs}$ for instance a store of a value in GPR r to a shared variable would then have the form:

  $$\mathrm{sw}\ r\ gp\ (4 \cdot o)_{16}$$

  Formally an access to a global shared variable by an instruction $I \in \mathbb{I}_{\mathrm{MASM}}$ is denoted by the predicate:

  $$gsv(I) \stackrel{def}{\equiv} mem(I) \wedge rs(I) = gp \wedge \exists o \in \mathit{off}_{gs}.\ imm(I) = (4 \cdot o)_{16}$$

Thus we define the $\mathcal{IO}$ predicate as follows:

$$S^n_{\mathrm{MASM}}.\mathcal{IO}(u,m,\varepsilon) = \mathrm{cas}(I(u,\lceil m\rceil)) \vee gsv(I(u,\lceil m\rceil))$$

For ownership-safe programs this implies that the addresses of the global shared variables should always be in set $\mathcal{S}$ (or be owned by the accessing unit). Ownership of other memory regions can then be obtained by using cas instructions and locks, or by synchronizing with the concurrent MASM computation units using lock-free algorithms based on the global shared variables.

- $S^n_{\mathrm{MASM}}.\mathcal{IP}$ — for easier verification of MASM code we could choose a coarse interleaving of MASM steps, i.e., have interleaving points before all $\mathcal{IO}$ steps. However, later we want to show how to use the *Cosmos* model order reduction theorem to allow for a simulation between S^n_{MASM} and S^n_{MIPS}. Therefore we choose the interleaving points to be in all MASM configurations where the consistency relation holds with some implementing MIPS execution. As we assume that the MASM assembler does not optimize across several MASM statements the simulation relation holds actually in *every* MASM configuration. Hence every MASM step is an interleaving point.

  $$S^n_{\mathrm{MASM}}.\mathcal{IP}(u,m,\varepsilon) = 1$$

  The more interesting instantiation are the interleaving-points of S^n_{MIPS} in this case. As we will see later (cf. Section 5.6.1) there we will have to choose the set of interleaving-point addresses (set A_{cp} in the MIPS *Cosmos* machine definition) in such a way that it contains all addresses in the code region where the assembled code of a MASM statement begins according to the adr function.

This finishes the definition of our concurrent MASM model. However we still need to discharge $insta_r(S^n_{\mathrm{MASM}})$ which is easy because in MASM there are no serial reads due to fetching, thus the $reads$-set is independent of memory.

PROOF: We have to prove the following statement for MASM core configuration $u \in \mathbb{K}_{\mathrm{MASM}}$ and memories $m, m' : S^n_{\mathrm{MASM}}.\mathcal{A} \to S^n_{\mathrm{MASM}}.\mathcal{V}$. Let $R = S^n_{\mathrm{MASM}}.reads(u,m,\varepsilon)$ in:

$$m|_R = m'|_R \implies S^n_{\mathrm{MASM}}.reads(u,m',\varepsilon) = R$$

By definition with $R' = S^n_{\mathrm{MASM}}.reads(u,m',\varepsilon)$ we have:

$$R = R(u.cpu, I(u,m)) \qquad R' = R(u.cpu, I(u,m'))$$

Before (cf. Section 4.2.1) we already observed from the defintion of $I(c_\mu)$ for $c_\mu = (u,m)$:

$$I(u,m) = u.\pi[u.s[|u.s|].p].body[u.s[|u.s|].loc]$$

Hence we see that the current instruction is not depending on the memory m. Therefore we have $I(u,m) = I(u,m')$ and $R' = R$ follows trivially. □

4.3 C Intermediate Language

The MASM language introduced in the previous section does not abstract very far from the underlying instruction set architecture. In order to instantiate our *Cosmos* model with a higher-level language, in this section we introduce semantics for a simplified version of C. We present the C Intermediate Language (C-IL) that was developed by S. Schmaltz [SS12] in order to justify the C verification approach applied in the Verisoft XT hypervisor verification project [Sch13b]. Here, for brevity, we do not give a full definition of the C-IL semantics and omit technical details like type and expression evaluation, that can be looked up in the original research documents. Instead we concentrate on the parts which are relevant for stating a compiler consistency relation and a simulation theorem between a C-IL computation and a MIPS implementation. Such a compiler consistency relation and simulation theorem was already stated by A. Shadrin for the VAMP ISA [Sha12] and we adapt it to the MIPS architecture defined above. Moreover we fix the architecture-dependent environment parameters of C-IL according to our MIPS model.

In what follows we first define the syntax and semantics of C-IL, then we establish a Compiler Consistency Relation between C-IL and a MIPS implementation. Finally we instantiate a *Cosmos* machine with the C-IL semantics obtaining a concurrent C-IL model. Applying the *Cosmos* order reduction theorem then allows to establish a model of structured parallel C-IL, where C program threads are interleaved only at volatile variable accesses.

4.3.1 C-IL Syntax and Semantics

As C-IL is an intermediate representation of C, C-IL programs are not written by a programmer but rather obtained from a C program by parsing and translation. This allows us to focus only on essential features of a high-level programming language and disregard a large portion of the "syntactic sugar" found in C. In essence a C-IL program consists of type declarations, variable declarations, and a function table. The body of each function contains C-IL statements which comprise variable assignments (that may make use of pointer arithmetic), label-based control-flow commands, and primitives for function calls and returns. Before we can give a mathematical definition of C-IL programs, we need to introduce C-IL types, values, and expressions. Moreover there are some environment parameters for C-IL program execution.

Environment Parameters

C-IL is not defined for a certain underlying architecture, nor a given class of programs, nor a particular compiler. Thus there are many features of the environment, e.g., the memory type, operator semantics, composite type layout, or global variable placement, that must be seen as a parameter to program execution. In [Sch13b] this information is collected in the environment parameter $\theta \in params_{\text{C-IL}}$. Here we do not list the

components of θ in their entirety. On the one hand we fix some of the environment parameters for our MIPS-based version of C-IL. This means in particular, that:

- the endianness of the underlying architecture is *little endian* ($\theta.endianness = \mathbf{little}$),
- pointers consist of 4 bytes, i.e., they are bit strings of length 32 ($\theta.size_{ptr} = 4$),
- we only consider one compiler intrinsic function[6] for executing the MIPS Compare-and-Swap instruction ($\theta.intrinsics$ to be defined later), and
- we only use the 32-bit primitive types and the empty type[7], because our MIPS architecture does not support memory accesses with byte or halfword granularity ($\theta.\mathbb{T}_P = \{\mathbf{i32}, \mathbf{u32}, \mathbf{void}\}$).

On the other hand, as we do not present the technical details of expession evaluation, a lot of the environment information is actually irrelevant to us. Still the dependency of certain functions on the environment parameters will be visible by taking θ as an argument. In such cases we will give explanations on what kind of environment parameters the functions are actually depending. Nevertheless we will not disclose all the technical details which can be found in [Sch13b].

For defining the C-IL values and transition function however we will need to refer to two environment parameters in θ specificly:

- a mapping $\theta.\mathcal{F}_{adr}$ from the set of function names to memory addresses. This compiler-dependent function is used to convert function pointers to bit strings and store them in memory, which can be useful for, e.g., setting up interrupt descriptor tables in MIPS-86 systems.
- a mapping $\theta.R_{\mathbf{extern}}$ which returns a C-IL state transition relation for external procedures who are declared but not implemented within the C-IL program. A call to such a function then results in a non-deterministic transition according to the transition relation.

A more detailed description of these components shall be given when they are used.

Types

In general C-IL is based on the following sets of names.

- $\mathbb{V}$ — names of variables
- $\mathbb{T}_C$ — names of composite (struct) types
- $\mathbb{F}$ — names fields in composite (struct) type variables

[6] According to [Sch13b], compiler intrinsics are pre-defined functions whose bodies are inlined into the program code instead of creating a new stack frame, when compiling function calls. Usually intrinsics are external functions that are implemented in assembly language.

[7] The empty type **void** is used as a return type for functions that do not return any value.

- $\mathbb{F}_{name}$ — names of functions

Then we allow the following types in C-IL.

Definition 39 (C-IL Types) *The set $\mathbb{T}$ of all possible C-IL types constructed inductively according to the case split below. For any $t \in \mathbb{T}$ one of the following conditions holds.*

- $t \in \{\mathbf{void}, \mathbf{i32}, \mathbf{u32}\}$ — *t is a primitive type*
- $\exists t' \in \mathbb{T}.\ t = \mathbf{ptr}(t')$ — *t is a pointer to a value of type t'*
- $\exists t' \in \mathbb{T}, n \in \mathbb{N}.\ t = \mathbf{array}(t', n)$ — *t is an array of values with type t'*
- $\exists t' \in \mathbb{T}, T \in (\mathbb{T} \setminus \{\mathbf{void}\})^*.\ t = \mathbf{funptr}(t', T)$ — *t is a function pointer to a function which takes a list of input parameters with non-empty types according to list T and returns a value with type t'*
- $\exists t_C \in \mathbb{T}_C.\ t = \mathbf{struct}\ t_C$ — *t is a composite type with name t_C*

For composite types we do not store the detailed structure but just the name of the struct. As we will see later, the field definitions for all structs is part of the C-IL program and can thus be looked up there during type evaluation. Moreover the environment information θ contains a parameter to determine the offsets of struct components in memory. However we did not formally introduce this parameter as we will not use it explicitely in the frame of this thesis.

Besides the types listed above we also have type qualifiers which give hints to the compiler how accesses to variables with a certain qualified type shall be compiled and what kind of optimizations can be applied.

Definition 40 (C-IL Type Qualifiers) *The set of C-IL type qualifiers is defined as follows:*

$$\mathbb{Q} \equiv \{\mathbf{volatile}, \mathbf{const}\}$$

The type qualifier **volatile** is used in the type declaration of a variable to denote that this variable can change its value independent of the program, therefore we also use the name *volatile* variables. In particular other processes in the system like concurrent threads, interrupt handlers or devices can also access such variables. Consequently the compiler has to take care when compiling accesses to volatile variables in order to make sure that updates are actually visible to the other processes, and that read accesses do not return stale data. In other words, the value of a volatile variable should always be consistent with its implementation in shared memory. This implies that all accesses to volatile variables must be implemented by atomic operations. Thus there are limitations on the kind of optimizations the compiler can possibly apply on volatile variable accesses.

On the other hand, variables that are declared with a **const** type qualifier (*constant variables*) are supposed to keep their value and never be modified. Thus the compiler can perform more aggressive optimizations on accesses to these variables.

We need to extend our type definition to *qualified types* because in non-primitive types we might have different qualifiers on the different levels of nesting.

Definition 41 (Qualified C-IL Types) *The set $\mathbb{T}_Q$ of all qualified types in C-IL is constructed inductively as follows. For $(q,t) \in \mathbb{T}_Q$ we have the following cases.*

- *The empty type is not qualified:* $(q,t) = (\emptyset, \mathbf{void})$
- *Qualified primitive type:* $q \subseteq \mathbb{Q} \wedge t \in \{\mathbf{i32}, \mathbf{u32}\}$
- *Qualified pointer type:* $q \subseteq \mathbb{Q} \wedge \exists t' \in \mathbb{T}_Q.\ t = \mathbf{ptr}(t')$
- *Qualified array type:* $q \subseteq \mathbb{Q} \wedge \exists t' \in \mathbb{T}_Q, n \in \mathbb{N}.\ t = \mathbf{array}(t', n)$
- *Qualified function pointer type:*

$$q \subseteq \mathbb{Q} \wedge \exists t' \in \mathbb{T}_Q, T \in (\mathbb{T}_Q \setminus \{(\emptyset, \mathbf{void})\})^*.\ t = (q, \mathbf{funptr}(t', T))$$

- *Qualified struct type:* $q \subseteq \mathbb{Q} \wedge \exists t_C \in \mathbb{T}_C.\ t = (q, \mathbf{struct}\ t_C)$

Thus qualified types are pairs of a set of qualifiers (which may be empty) and a type which may be constructed using other qualified types. For qualified struct types, again, the qualified component declaration will be given elsewhere. Before we can define the C-IL values we need three more shorthands to determine the class of a type $t \in \mathbb{T}$.

$$\begin{aligned} isptr(t) &\stackrel{def}{\equiv} \exists t'.\ t = \mathbf{ptr}(t') \\ isarray(t) &\stackrel{def}{\equiv} \exists t', n.\ t = \mathbf{array}(t', n) \\ isfunptr(t) &\stackrel{def}{\equiv} \exists t', T.\ t = \mathbf{funptr}(t', T) \end{aligned}$$

Values

In this section we define sets of values for variables of the different C-IL types defined above. Note that the possible values of a variable do not depend on type qualifiers. A qualified type can be converted into an unqualified type by recursively removing the set of qualifiers leaving only the type information. Let this be done by the following function:

$$qt2t : \mathbb{T}_Q \to \mathbb{T}$$

Definition 42 (Primitive Values) *We define the set $val_{\mathbf{prim}}$ which contains all values for variables of primitive type.*

$$val_{\mathbf{prim}} \equiv \bigcup_{b \in \mathbb{B}^{32}} \{\mathbf{val}(b, \mathbf{i32}), \mathbf{val}(b, \mathbf{u32})\}$$

Primitive values consist of the constructor **val** and a 32 bit string as well as the type information whether that bit string should be interpreted as a signed (two's complement) or unsigned binary number. Note that this definition is a simplified version of the corresponding definition in [Sch13b] since we only need to consider 32 bit values in our MIPS-based C-IL semantics. Observe also, that we do not define a set of values for the primitive type **void** because this type is used to denote that no value is returned by a function. Consequently in C-IL we cannot evaluate values of type **void**.

Definition 43 (Pointer and Array Values) *The set $val_{\mathbf{ptr}}$ of values for pointers and arrays is defined as follows.*

$$val_{\mathbf{ptr}} \equiv \bigcup_{t \in \mathbb{T} \wedge (isptr(t) \vee isarray(t))} \{\mathbf{val}(a,t) \mid a \in \mathbb{B}^{32}\}$$

Here we merge the values for pointers and arrays because we treat array variables as pointers to the first element of an array. Accesses to fields of an array are then resolved via pointer arithmetic in expression evaluation. References to components of local variables of a function are represented by the following values.

Definition 44 (Local Variable Reference Values) *Let $\mathbb{V}$ be the set of all variable names, then the set of values for local variable references is defined as follows.*

$$val_{\mathbf{lref}} \equiv \bigcup_{t \in \mathbb{T} \wedge (isptr(t) \vee isarray(t))} \{\mathbf{lref}((v,o),i,t) \mid v \in \mathbb{V} \wedge o,i \in \mathbb{N}_0\}$$

Local variables are modeled as small separate memories, i.e., lists of bytes, to allow for pointer arithmetic on them. Therefore in order to refer to a component of a local variable the variable name v and the component's byte offset o are saved in the **lref** value. Moreover one needs to know the type t of the referenced component and the index of the function frame i in which the local variable is contained.

Concerning function pointers we distinguish between two kind of values. The first kind $val_{\mathbf{fptr}}$ is used for pointers to those functions of which we know the corresponding memory address according to $\theta.\mathcal{F}_{adr} : \mathbb{F}_{name} \rightharpoonup \mathbb{B}^{32}$. These function pointers can be stored in memory. For function pointers to other functions $f \in \mathcal{F}_{name}$ where $\mathcal{F}_{adr}(f) = \bot$ we use symbolic values from the set val_{fun}. Such pointers cannot be stored in memory but only be dereferenced, resulting in a call of the referenced function.

Definition 45 (Function Pointer Values) *The two sets of values $val_{\mathbf{fptr}}$ and $val_{\mathbf{fun}}$ for C-IL function pointers are defined as follows.*

$$\begin{aligned} val_{\mathbf{fptr}} &\equiv \bigcup_{t \in \mathbb{T} \wedge isfunptr(t)} \{\mathbf{val}(a,t) \mid a \in \mathbb{B}^{32}\} \\ val_{\mathbf{fun}} &\equiv \{\mathbf{fun}(f,t) \mid f \in \mathbb{F}_{name} \wedge isfunptr(t)\} \end{aligned}$$

Finally the set val of all C-IL values is the union of primitive, pointer, local variable reference, and function pointer types.

$$val \stackrel{def}{\equiv} val_{\mathbf{prim}} \cup val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup val_{\mathbf{fptr}} \cup val_{\mathbf{fun}}$$

Note that we do not have values for structs, since we can only evaluate the components of struct variables but not the complete struct.

Expressions

Expressions in C-IL are used on the left and right side of variable assignments, as conditions in control-flow statements, as function identifiers and input parameters, to determine the variable where to store the returned value of a function, as well as the return value itself. A successful evaluation of an expression returns a values from **val**. Thus expressions encode primitive values, pointers, local variable references and function pointers. In C-IL expressions we can use the following unary mathematical operators from the set $\mathbb{O}_1$ for arithmetic, binary, and logical negation.

$$\mathbb{O}_1 \stackrel{def}{\equiv} \{-, \sim, !\}$$

The set $\mathbb{O}_2$ comprises all available binary mathematical operators.

$$\mathbb{O}_2 \stackrel{def}{\equiv} \{+, -, *, /, \%, <<, >>, <, >, <=, >=, ==, !=, \&, |, \hat{}, \&\&, ||\}$$

From left to right these symbols represent addition, subtraction[8], multiplication, integer division, modulo, binary left shift, right shift, less, greater, less or equal, greater or equal, equal, unequal, binary AND, OR, and XOR, as well as logical AND, and OR. Other C operators, e.g., for taking the address of a variable or pointer-dereferencing, are not considered mathematical operators. They are treated in the following definition of the structure of C-IL expressions.

Definition 46 (C-IL Expression) *The set $\mathbb{E}$ contains all possible C-IL expressions and is defined inductively. Every $e \in \mathbb{E}$ obeys one of the following rules.*

- $e \in val$ *— e is a constant C-IL value.*
- $e \in \mathbb{V}$ *— e identifies a C-IL variable by its name. In expression evaluation local variables take precedence over global variables with the same name.*
- $e \in \mathbb{F}_{name}$ *— e identifies a C-IL function by its name. Such expressions are used both for calling a function as well as creating function pointers.*
- $\exists e' \in \mathbb{E}, \ominus \in \mathbb{O}_1.\ e \equiv \ominus e'$ *— e is obtained by applying a unary operator on another expression.*
- $\exists e', e'' \in \mathbb{E}, \oplus \in \mathbb{O}_2.\ e \equiv (e' \oplus e'')$ *— e is obtained by combining two other expressions with a binary operator.*
- $\exists c, e', e'' \in \mathbb{E}.\ e \equiv (c\ ?\ e' : e'')$ *— e consists of three sub-expressions that are combined using the ternary conditional operator. If c evaluates to a value other than zero, then e evaluates to the value of e', otherwise the value of e'' is returned.*
- $\exists e' \in \mathbb{E}, t \in \mathbb{T}_Q.\ e \equiv (t)e'$ *— e represents a type cast of expression e' to qualified type t*

[8] Note that the same symbol is used for unary and binary minus, however in the definition of expressions they are used unambiguously.

- $\exists e' \in \mathbb{E}.\ e \equiv *(e')$ — *e is the value obtained from dereferencing the pointer that is encoded by expression e'*
- $\exists e' \in \mathbb{E}.\ e \equiv \&(e')$ — *e is the address of the sub-variable denoted by expression e'. Sub-variables are either variables or components of variables.*
- $\exists e' \in \mathbb{E}, f \in \mathbb{F}.\ e \equiv (e').f$ — *e represents the component with field name f of a struct-type variable described by expression e'*
- $\exists t \in \mathbb{T}_Q.\ e \equiv \mathbf{sizeof}(t)$ — *e evaluates to the size in bytes of a variable with type t.*
- $\exists e' \in \mathbb{E}.\ e \equiv \mathbf{sizeof}(e')$ — *e evaluates to the size in bytes of the type of expression e'.*

Note that not all expressions that can be constructed using this scheme are meaningful. For instance, an expression $e \in \mathbb{V}$ might reference a variable that does not exist, or an expression e' in $e \equiv \&(e')$ might encode a constant instead of a sub-variable. The well-formedness of expressions is checked during expression evaluation.

Note moreover that $\mathbb{E}$ does not provide a dedicated operation for accessing fields of array variables. This is because the common notation $a[i]$ for accessing field i of an array variable a is just syntactic sugar for the expression $*((a + i))$. Similarly if a is a pointer to a struct-type variable then the common shorthand $a \rightarrow f$ for accessing field f of the referenced struct can be represented by the expression $(*(a)).f$.

Programs

Before we can define the structure of C-IL programs we need to introduce the statements of the C Intermediate Language.

Definition 47 (C-IL Statements) *The set $\mathbb{I}_{\text{C-IL}}$ contains all C-IL statements and is defined inductively. For $s \in \mathbb{I}_{\text{C-IL}}$ we have the following cases.*

- $\exists e, e' \in \mathbb{E}.\ s \equiv (e = e')$ — *s is an assignment of the value encoded by expression e' to the sub-variable or memory location represented by expression e.*
- $\exists l \in \mathbb{N}.\ s \equiv \mathbf{goto}\ l$ — *s is a goto statement which redirects control-flow to label l in the current function.*
- $\exists e \in \mathbb{E}, l \in \mathbb{N}.\ s \equiv \mathbf{ifnez}\ e\ \mathbf{goto}\ l$ — *s is a conditional goto statement which redirects control-flow to label l in the current function if e evaluates to a non-zero value.*
- $\exists e, e' \in \mathbb{E}, E \in \mathbb{E}^*.\ s \equiv (e' = \mathbf{call}\ e(E))$ — *s represents a function call to the function identified by expression e (which must evaluate to a function pointer value), passing the input parameters according to expression list E. The value returned by the function is assigned to the sub-variable of memory location identified by expression e'.*
- $\exists e \in \mathbb{E}, E \in \mathbb{E}^*.\ s \equiv \mathbf{call}\ e(E)$ — *s is a function call without return value.*

- $\exists e \in \mathbb{E}.\ s \equiv \mathbf{return}\ e$ — *s is a return statement. Executing s returns from the current function with the return value denoted by expression e.*
- $s \equiv \mathbf{return}$ — *s is a return statement without return value. This variant is used for functions with return type* **void**.

Note that above we renamed the set of statements $\mathbb{S}$ from [Sch13b] to $\mathbb{I}_{\text{C-IL}}$ in order to avoid collision with our set $\mathbb{S}$ of *Cosmos* machine signatures. The statements listed above make up the body of every C-IL function. All relevant information about the particular functions of a C-IL program are stored in a function table.

Definition 48 (C-IL Function Table Entry) *The function table entry fte of a C-IL function has the following structure of type $FunT$.*

$$fte = (rettype, npar, V, P) \in FunT$$

Here the components of fte have the following meaning:

- $rettype \in \mathbb{T}_Q$ — *the type of the function's return value (return type)*
- $npar \in \mathbb{N}$ — *the number of input parameters for the function*
- $V \in (\mathbb{V} \times \mathbb{T}_Q)^*$ — *a list of parameter and local variable declarations containing pairs of variable name and type, where the first $npar$ entries represent the input parameters*
- $P \in \mathbb{I}^*_{\text{C-IL}} \cup \{\mathbf{extern}\}$ — *either the function body containing a list of C-IL statements that will be executed upon function call, or the keyword* **extern** *which marks the function as an external function. The effect of external functions is specified by the environment parameter $\theta.R_{\mathbf{extern}}$.*

Again, we renamed components $\mathcal{V}$ and $\mathcal{P}$ from [Sch13b] to V and P here because of definitions with the same name in this thesis. Now a C-IL program is defined as follows.

Definition 49 (C-IL Program) *A C-IL program π has type $prog_{\text{C-IL}}$*

$$\pi = (V_G, T_F, \mathcal{F}) \in prog_{\text{C-IL}}$$

with components:

- $V_G \in (\mathbb{V} \times \mathbb{T}_Q)^*$ — *declaration of global variables*
- $T_F : T_C \rightharpoonup (\mathbb{F} \times \mathbb{T}_Q)^*$ — *a type table for struct types, containing the type for every field of a given struct type name*
- $\mathcal{F} : \mathbb{F}_{name} \rightharpoonup FunT$ — *the function table, containing the function table entries for all functions of the program*

Because we have the type table $\pi.T_F$ for struct types t we can represent a struct with name t_C simply by the construction $t = \mathbf{struct}\ t_C$ without need to save the concrete structure of the struct in the type. This is useful to break the cyclical definitions in many common data structures which may contain pointers to variables of their own type. For instance in linked lists, a list item usually contains a pointer to the next list item. Instead of having a cyclical definition like

$$t_{list} \equiv \mathbf{struct}((v, \mathbf{i32}) \circ (next, \mathbf{ptr}(t_{list})))$$

one can then separately define the name and structure of the list item type:

$$t_{list} \equiv \mathbf{struct}\ item \qquad \pi.T_F(item) = (v, \mathbf{i32}) \circ (next, \mathbf{ptr}(t_{list}))$$

Naturally there are a lot of well-formedness conditions on C-IL programs, for instance, that only declared sub-variables may be used, or that **goto** statements may only target labels within the bounds of the current function. We could define all these static checks on the C-IL program separately as in the MASM model. However, in [Sch13b] most of the possible faults in a C-IL program are captured as run-time errors during type evaluation, expression evaluation, and application of the C-IL transition function. However, there are a few conditions missing concerning control-flow statements. We introduce the following predicate to denote that $s \in \mathbb{I}_{\text{C-IL}}$ is a C-IL control-flow statement which is targeting a label $l \in \mathbb{N}$.

$$ctrl(s, l) \stackrel{def}{\equiv} s = \mathbf{goto}\ l \vee \exists e \in \mathbb{E}.\ s = \mathbf{ifnez}\ e\ \mathbf{goto}\ l$$

Now we can define the well-formedness conditions on C-IL programs that are not covered by the run-time error definitions of [Sch13b] which will be given later.

Definition 50 (C-IL Program Well-formedness) *We consider a C-IL program π to be well-formed if it obeys the conditions that (i) any control-flow instruction targets only labels within the corresponding function and that (ii) only functions with return type* **void** *omit returning a value.*

$$\begin{aligned} wfprog_{\text{C-IL}}(\pi) \stackrel{def}{\equiv}\ & \forall f \in \mathbb{F}_{name}, s \in \mathbb{I}_{\text{C-IL}}.\ \pi.\mathcal{F}(f) \neq \bot \wedge s \in \pi.\mathcal{F}(f).P \implies \\ & \textit{(i)}\quad ctrl(s, l) \implies l \in [1 : |\pi.\mathcal{F}(f).P|] \\ & \textit{(ii)}\quad s = \mathbf{return} \implies \pi.\mathcal{F}(f).rettype = \mathbf{void} \end{aligned}$$

Note that according to [Sch13b] it is allowed to use a statement **return** e for some expression $e \in \mathbb{E}$ to return from a function with return type **void**. The returned value is simply ignored then.

Configurations

Finally we can define the configurations of the C-IL model. Basically a C-IL configuration consists of a global memory and the current state of the stack. The stack models contains all the local information that is needed for the execution of C-IL functions. For every new function call a stack frame with the following structure is put on the stack.

Definition 51 (C-IL Stack Frame) *A C-IL stack frame s is a record*

$$s = (\mathcal{M}_\mathcal{E}, rds, f, loc) \in frame_{\text{C-IL}}$$

containing the components:

- $f \in \mathbb{F}_{name}$ *— the name of the function, to which the stack frame belongs.*
- $\mathcal{M}_\mathcal{E} : \mathbb{V} \rightharpoonup (\mathbb{B}^8)^*$ *— the memory for local variables and parameters. The content of a local variable or parameter is represented as a list of bytes, thus allowing for pointer arithmetic within the variables.*
- $rds \in val_{\mathbf{ptr}} \cup val_{\mathbf{lref}} \cup \{\bot\}$ *— the return destination for function calls from f, which contains a reference to the sub-variable where to store the return value of a called function. If the called function has return type* **void** *we set rds to $\bot$.*
- $loc \in \mathbb{N}$ *— the location counter, indexing the next statement in the function body of f to be executed.*

Then the definition of a C-IL configuration is straight-forward.

Definition 52 (C-IL Configuration) *A C-IL configuration c is a record*

$$c = (\mathcal{M}, s) \in conf_{\text{C-IL}}$$

containing the components:

- $\mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^8$ *— the global byte-adressable memory*
- $s \in frame^*_{\text{C-IL}}$ *— the C-IL stack, containing C-IL stack frames, where the top frame is at the end*[9] *of the list.*

See Fig. 18 for an illustration. In most cases the execution of a step of a C-IL program depends only on the top-most frame of the stack and the memory. The location pointer of the stack frame points to the statement in the corresponding function's body that shall be executed next. Global variables are located in the global memory, moreover there are the local variables and parameters contained in the local memory of each stack frame. Local variables and parameters obscure global variables with the same name. By using variable identifiers one can only access global memory and the local memory of the top-most frame, however using local references one can also update local memories of the lower frames in the stack. The flat byte-addressable global memory can also be accessed by dereferencing plain memory addresses (pointer-arithmetic). Note however that, while in fact the stack is implemented in a part of the global memory, C-IL does not allow to access local variables or other components of the stack via pointer-arithmetic. Location and layout of stack frames is undisclosed and in the simulation theorem we will have software conditions prohibiting explicit memory accesses to the stack region.

[9] Here we differ from [Sch13b] where the top frame is the head of $c.s$.

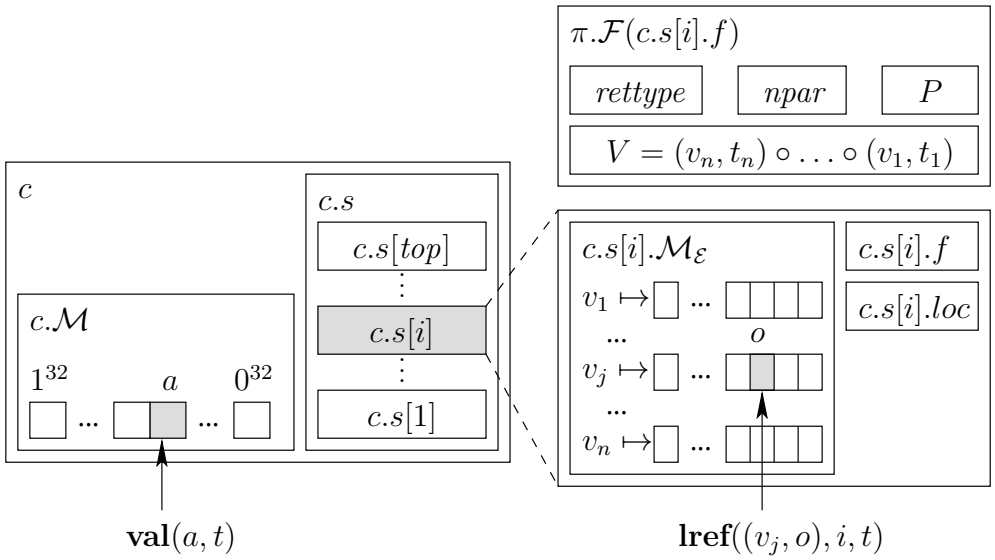

Figure 18: Illustration of the C-IL configurations and where pointers and local references are pointing to. This figure is copied from [Sch13b] and adapted to our setting and notation. In particular, here $top \equiv |c.s|$.

An important part in the execution of C-IL steps is the evaluation of types and expressions, where the first is useful to define the second. However the detailed definitions of the evaluation functions are quite technical. They can be found in Section 5.8 of [Sch13b]. Here we only declare and use them.

Definition 53 (C-IL Type and Expression Evaluation) *We introduce the C-IL type evaluation function ${\tau_Q}_c^{\pi,\theta}$ which returns the type for a given C-IL expression wrt. C-IL configuration c, program π, and environment parameters θ.*

$$\tau_{Q}^{\cdot,\cdot}(\cdot) : conf_{\text{C-IL}} \times prog_{\text{C-IL}} \times params_{\text{C-IL}} \times \mathbb{E} \rightharpoonup \mathbb{T}_Q$$

Similarly, we introduce the C-IL expression evaluation function $[\![\cdot]\!]_c^{\pi,\theta}$ which returns the for a given C-IL expression.

$$[\![\cdot]\!]^{\cdot,\cdot} : conf_{\text{C-IL}} \times prog_{\text{C-IL}} \times params_{\text{C-IL}} \times \mathbb{E} \rightharpoonup val$$

Both functions are defined by structural induction on the given expression. They are partial functions because not all expressions are well-formed and can be evaluated properly for a given program and C-IL configuration. In such cases the type and value of an expression e are undefined and we have ${\tau_Q}_c^{\pi,\theta}(e) = \bot$, and $[\![e]\!]_c^{\pi,\theta} = \bot$ respectively.

A typical case of an erroneous expression is a reference to a variable name that is not declared. The type evaluation of a pointer dereferencing $*(e)$ fails if e is not of the type pointer or array. Similarly, the type of an address $\&(e)$ of an expression e is only defined if e describes a sub-variable or a dereferenced pointer.

In expression evaluation we have the same restrictions as above, i.e., referencing undeclared variables or function names results in an undefined value. For dereferencing a pointer there is a case distinction on the type of the pointer, thus if the type of some expression $*(e)$ is undefined so is its value. In the evaluation it is distinguished whether the pointer points to a primitive value or to an array. In the first case one simply reads the referenced memory address, in the latter case the array itself is returned. We cannot reference or dereference complete struct-type variables, but only their primitive or array fields.

The evaluation functions depend on the environment parameters for several reasons. First the type returned by the **sizeof** operator is defined in θ. In expression evaluation one has to know the offset of the fields in the memory representation of composite variables, which is also a compiler-dependent environment parameter. For evaluating function pointers we need to check $\theta.\mathcal{F}_{adr}$ in order to determine which of the two function pointer values should be used. Moreover the effects of mathematical operators and type casts are also compiler-dependent.

Before we can define the C-IL transition function we need to make a few more definitions. Up to now we have not defined the size of types in memory. It is computed by a function

$$size_\theta : \mathbb{T} \to \mathbb{N}$$

which returns the number of bytes occupied in memory by a value of a given type. Its definition is based on θ because the layout of struct types in memory is depending on the compiler. However for primitive or pointer types t we have $size_\theta(t) = 4$, as expected. Moreover if t is an array of n elements with type t' then $size_\theta(t) = n \cdot size_\theta(t')$. The complete definition can be found in [Sch13b]. Using the type evaluation and type size functions we can define the following well-formedness conditions for C-IL configurations similar to the ones in our MASM model.

Definition 54 (Well-formedness of C-IL Configurations) *To be well-formed we require for any configuration $c \in conf_{\text{C-IL}}$, program $\pi \in prog_{\text{C-IL}}$, and environment parameter $\theta \in params_{\text{C-IL}}$ that in every stack frame (i) the current function is defined, (ii) the sizes of local memories correspond to the variable declarations of the corresponding functions, (iii) below the top frame the type of the return destination agrees with the return type of the called function in the frame above (with higher index), and (iv) the current location never leaves the function body. Moreover (v) the program is well-formed. Given a stack $s \in frame^*_{\text{C-IL}}$ we first define:*

$$\begin{aligned}
wfs_{\text{C-IL}}(s,\pi,\theta) \;\equiv\; & \forall i \in [1:|s|]. \\
& \textit{(i)} \quad \pi.\mathcal{F}(s[i].f) \neq \bot \\
& \textit{(ii)} \quad \forall (v,t) \in \pi.\mathcal{F}(s[i].f).V \implies \\
& \qquad\qquad s[i].\mathcal{M}_\mathcal{E}(v) \neq \bot \wedge |s[i].\mathcal{M}_\mathcal{E}(v)| = size_\theta(qt2t(t)) \\
& \textit{(iii)} \quad i < |s| \implies \tau_{Q_c}^{\pi,\theta}(s[i].rds) = \pi.\mathcal{F}(s[i+1].f).rettype \\
& \textit{(iv)} \quad s[i].loc \in [1:|\pi.\mathcal{F}(s[i].f)|]
\end{aligned}$$

and set $wf_{\text{C-IL}}(c,\pi,\theta) \equiv wfprog_{\text{C-IL}}(\pi) \wedge wfs_{\text{C-IL}}(c.s,\pi,\theta)$ according to (v).

Thus the well-formedness of C-IL configurations depends only on the stack but not on the global memory. As we have introduced C-IL configurations we can also complete the definition of the environment parameter $\theta.R_{\mathbf{extern}}$.

Definition 55 (External Procedure Transition Relations) *We use the environment parameter $\theta.R_{\mathbf{extern}}$ to define the effect of external procedures whose implementation is not given by the C-IL programs. It has the following type*

$$\theta.R_{\mathbf{extern}} : \mathbb{F}_{name} \rightharpoonup 2^{val^* \times conf_{\text{C-IL}} \times conf_{\text{C-IL}}}$$

where for an external procedure x, such that $\pi.\mathcal{F}(x).P = \mathbf{extern}$, the set $\theta.R_{\mathbf{extern}}(x)$ contains tuples $((i_1, \ldots, i_n), c, c')$ with the components:

- *$i_1, \ldots, i_n$ — the input parameters to the external procedure*
- *c, c' — the pre- and post state of the transition*

In case an external procedure x is called with a list of input parameters from a C-IL configuration c, the next configuration c' is determined by non-deterministically choosing a fitting transition from $\theta.R_{\mathbf{extern}}(x)$.

Closely related to external procedures are the compiler intrinsic functions that are defined by $\theta.intrinsics : \mathbb{F}_{name} \rightharpoonup FunT$. Intrinsics are predefined functions that are provided by the compiler to the programmer, usually to access certain system resources that are not visible in pure C. As anounced before the only intrinsic function considered in our scenario is cas, which is a wrapper function for the Compare-and-Swap assembly instruction. We define $\theta.intrinsics(cas) = fte_{cas}$ below. For all other function names $f \neq cas$ we have $\theta.intrinsics(f) = \bot$.

$$\begin{aligned} fte_{cas}.rettype &= (\emptyset, \mathbf{void}) \\ fte_{cas}.npar &= 4 \\ fte_{cas}.V &= (a, (\emptyset, \mathbf{ptr}(\{\mathbf{volatile}\}, \mathbf{i32}))) \circ (u, (\emptyset, \mathbf{i32})) \\ &\quad \circ (v, (\emptyset, \mathbf{i32})) \circ (r, (\emptyset, \mathbf{ptr}(\emptyset, \mathbf{i32}))) \\ fte_{cas}.P &= \mathbf{extern} \end{aligned}$$

As most intrinsic functions cas is implemented in assembly and thus an external function. It takes 4 input arguments a,u,v, and r, where a is a pointer to the volatile memory location that shall be swapped, u is the value with which the memory location referenced by a is compared, and v is the value to be swapped in. The content of the memory location pointed to by a is written to the subvariable referenced by the fourth parameter[10] r. Since the intrinsics are provided by the compiler they are not part of the program-based function table. We define the combined function table $\mathcal{F}_\pi^\theta$ as follows.

$$\mathcal{F}_\pi^\theta \stackrel{def}{=} \pi.\mathcal{F} \uplus \theta.intrinsics$$

[10]Note that the cas instruction of the MIPS ISA has only three parameters. Thus in the implementation of cas an additional write instruction is needed to update the memory location referenced by r.

Knowing the semantics of the cas instruction of MIPS, we would also like to define the external procedure transition relation $R_{\mathbf{extern}}(cas)$. However we first need some more notation for updating a C-IL configuration. When writing C-IL values to the global or some local memory, they have to be broken down into sequences of bytes. First we need a function $bits2bytes : \mathbb{B}^{8n} \to (\mathbb{B}^8)^n$ convert a bit string whose length is a multiple of 8 into a byte string.

$$bits2bytes(x[m:0]) \stackrel{def}{\equiv} \begin{cases} bits2bytes(x[m:8]) \circ (x[7:0]) & : \quad m > 7 \\ (x[m:0]) & : \quad \text{otherwise} \end{cases}$$

Then the conversion from C-IL values to bytes is done by the following partial function.

$$val2bytes_\theta(v) \stackrel{def}{\equiv} \begin{cases} bits2bytes(b) & : \quad v = \mathbf{val}(b,t) \\ \bot & : \quad \text{otherwise} \end{cases}$$

Note that this definition excludes local variable references and symbolic function pointers. For these values the semantics does not provide a binary representation because for local subvariables and functions f where $\theta.\mathcal{F}_{adr}(f) = \bot$, the location in memory is unknown. Note also that $val2bytes_\theta$ depends on the environment parameter θ because the conversion to byte strings is depending on the endianness of the underlying memory system. As our MIPS ISA uses little endian memory representations we simplified the definition of $val2bytes_\theta$ which actually contains a case distinction on $\theta.endianness$ in [Sch13b]. We still keep the θ though, to keep the notation consistent.

Now we can introduce helper functions to write the global and local memories of a C-IL configuration. We copy the following three definitions literally from Section 5.7.1 of [Sch13b], with the single modifcation that we fix the pointer size to 4 bytes, and adapt the indexing of the local memories to start with 1 instead of 0.

Definition 56 (Writing Byte-Strings to Global Memory) *We define the function*

$$write_{\mathcal{M}} : (\mathbb{B}^{32} \to \mathbb{B}^8) \times \mathbb{B}^{32} \times (\mathbb{B}^8)^* \to (\mathbb{B}^{32} \to \mathbb{B}^8)$$

that writes a byte-string B to a global memory $\mathcal{M}$ starting at address a such that

$$\forall x \in \mathbb{B}^{32}.\ write_{\mathcal{M}}(\mathcal{M},a,B)(x) = \begin{cases} \mathcal{M}(x) & \langle x \rangle - \langle a \rangle \notin \{0,\ldots,|B|-1\} \\ B[\langle x \rangle - \langle a \rangle] & \text{otherwise} \end{cases}$$

Definition 57 (Writing Byte-Strings to Local Memory) *We define the function*

$$write_{\mathcal{E}} : (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*) \times \mathbb{V} \times \mathbb{N}_0 \times \mathbb{B}^8 \rightharpoonup (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*)$$

that writes a byte-string B to variable v of a local memory $\mathcal{M}_{\mathcal{E}}$ starting at offset o such that

$$\forall w \in \mathbb{V}, i \in [1 : |\mathcal{M}_{\mathcal{E}}(w)|].$$
$$write_{\mathcal{E}}(\mathcal{M}_{\mathcal{E}},v,o,B)(w)[i] = \begin{cases} \mathcal{M}_{\mathcal{E}}(w)[i] & w \neq v \vee i \notin \{o+1,\ldots,o+|B|\} \\ B[i-o] & \text{otherwise} \end{cases}$$

If, however, $|B| + o > |\mathcal{M}_{\mathcal{E}}(v)|$ or $v \notin \mathrm{dom}(\mathcal{M}_{\mathcal{E}})$, the function is undefined for the given parameters.

Definition 58 (Writing a Value to a C-IL Configuration) *We define the function*

$$write : params_{\text{C-IL}} \times conf_{\text{C-IL}} \times val \times val \rightharpoonup conf_{\text{C-IL}}$$

that writes a given C-IL value y to a C-IL configuration c at the memory pointed to by pointer x according to environment parameters θ as

$$write(\theta,c,x,y) \stackrel{def}{=} \begin{cases} c[\mathcal{M} := write_{\mathcal{M}}(c.\mathcal{M}, a, val2bytes_\theta(y))] & : x = \mathbf{val}(a, \mathbf{ptr}(t)) \\ & \quad \wedge\, y = \mathbf{val}(b, t) \\ c' & : x = \mathbf{lref}((v, o), i, \mathbf{ptr}(t)) \\ & \quad \wedge\, y = \mathbf{val}(b, t) \\ write(\theta, c, \mathbf{val}(a, \mathbf{ptr}(t)), y) & : x = \mathbf{val}(a, \mathbf{array}(t, n)) \\ write(\theta, c, \mathbf{lref}((v, o), i, \mathbf{ptr}(t)), y) & : x = \mathbf{lref}((v, o), i, \mathbf{array}(t, n)) \\ \text{undefined} & : \text{otherwise} \end{cases}$$

where $c'.s[i].\mathcal{M}_\mathcal{E} = write_\mathcal{E}(c.s[i].\mathcal{M}_\mathcal{E}, v, o, val2bytes_\theta(y))]$ and all other parts of c' are identical to c.

In the first case x contains a pointer to some value in global memory of type t and we are overwriting it with the primitive or pointer value y. When x is a local variable reference, we update the referenced variable with y in the specified local memory, starting at the given offset. Since arrays in C are treated as pointers to the first element of the array, any write operation to an array is transformed accordingly. Observe that $write$ checks for type safety, i.e., that value y and the write target specified by x have the same type. Moreover we cannot update c using symbolic function pointers for x, because these pointers are not associated with any resource in c.

We also provide a function $inc_{loc} : conf_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}}$ to increment the location counter of the top stack frame. It is undefined if the stack is empty, otherwise:

$$inc_{loc}(c) = c[s := c.s[top \mapsto (c.s[top])[loc := c.s[top].loc + 1]]]$$

Now we can define the transition relation $\theta.R_{\mathbf{extern}}(cas)$ for the cas compiler intrinsic function. It consists of two subrelations depending on whether the comparison in the Compare-and-Swap instruction was successful or not. In the first case we have:

$$\begin{aligned} \rho_{cas}^{swap} \stackrel{def}{=} \{((a, u, v, r), c, c''') \mid\ & \exists x, b \in \mathbb{B}^{32}.\ a = \mathbf{val}(x, \mathbf{ptr}(\mathbf{i32})) \wedge u = \mathbf{val}(b, \mathbf{i32}) \\ & \wedge\, c.\mathcal{M}_4(x) = b \wedge \exists c', c'' \in conf_{\text{C-IL}}.\ c' = write(\theta, c, a, v) \\ & \wedge\, c'' = write(\theta, c', r, \mathbf{val}(c.\mathcal{M}_4(x), \mathbf{i32})) \wedge\ c''' = inc_{loc}(c'') \} \end{aligned}$$

The memory location pointed to by a equals the test value u, consequently it is updated with v and its old value is stored in r. In addition the current location in the top frame

is incremented. For the fail case when u does not equal the referenced value of a, the memory location is not updated. The rest of the transition is identical to the case above.

$$\begin{aligned}\rho_{cas}^{fail} \stackrel{def}{=} \{((a,u,v,r),c,c'') \mid\ & \exists x, b \in \mathbb{B}^{32}.\ a = \mathbf{val}(x, \mathbf{ptr}(\mathbf{i32})) \wedge u = \mathbf{val}(b, \mathbf{i32}) \\ & \wedge\ c.\mathcal{M}_4(x) \neq b \wedge \exists c', c'' \in conf_{\text{C-IL}}.\ c'' = inc_{loc}(c') \\ & \wedge\ c' = write(\theta, c, r, \mathbf{val}(c.\mathcal{M}_4(x), \mathbf{i32}))\}\end{aligned}$$

Of course the overall transition relation is the disjunction of both cases.

$$\theta.R_{\mathbf{extern}}(cas) \ = \ \rho_{cas}^{swap} \cup \rho_{cas}^{fail}$$

Before we can define the C-IL transition function there are two more helper functions left to introduce. The first function $\tau : \mathbb{V} \to \mathbb{T}$ derives the type of values.

$$\tau(x) \stackrel{def}{=} \begin{cases} t & : \quad x = \mathbf{val}(y,t) \\ t & : \quad x = \mathbf{fun}(f,t) \\ t & : \quad x = \mathbf{lref}((v,o),i,t) \end{cases}$$

Last we define function $zero(\theta, x) : params_{\text{C-IL}} \times \mathbb{V} \rightharpoonup \mathbb{B}$ which checks whether a given primitive or pointer value equals zero.

$$zero(\theta, x) \stackrel{def}{=} \begin{cases} (a = 0^{8 \cdot size_\theta(t)}) & : \quad x = \mathbf{val}(a,t) \\ \bot & : \quad \text{otherwise} \end{cases}$$

We finish the introduction of the C-IL semantics with the definition of the C-IL transition function in the next sub-section. It is in great portions a literal copy of Section 5.8.3 in [Sch13b]. Apart from editorial changes and adaptations to our indexing of the stack, we only modify the C-IL input alphabet.

Transition Function

For given C-IL program π and environment parameters θ, we define a partial transition function

$$\delta_{\text{C-IL}}^{\pi,\theta} : conf_{\text{C-IL}} \times \Sigma_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}}$$

where $\Sigma_{\text{C-IL}}$ is an input alphabet used to resolve non-deterministic choice occurring in C-IL semantics. In fact, there are only two kinds of non-deterministic choice in C-IL: the first occurs in a function call step – the values of local variables of the new stack frame are not initialized, thus, they are chosen arbitrarily; the second is due to the possible non-deterministic nature of external function calls – here, one of the possible transitions specified by relation $\theta.R_{\mathbf{extern}}$ is chosen. To resolve these non-deterministic choices, our transition function gets as an input $in = (\mathcal{M}_{lvar}, \eta) \in \Sigma_{\text{C-IL}}$ containing a mapping $\mathcal{M}_{lvar}$ of function names to local memory configurations, and a mapping η of transition functions for computing the result of external function calls.

$$\Sigma_{\text{C-IL}} \ \subseteq \ (\mathbb{F}_{name} \rightharpoonup (\mathbb{V} \rightharpoonup (\mathbb{B}^8)^*)) \times (\mathbb{F}_{name} \rightharpoonup (val^* \times conf_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}}))$$

This setting deviates from [Sch13b] where the inputs are either a single local memory configuration $\mathcal{M}_{lvar}$, or an updated C-IL configuration c', or a symbolic value $\perp$ to denote deterministic steps. However this opens up the possibility of run-time errors due to nonsensical input sequences which do not provide the right inputs for the current statement, e.g., a $\perp$ symbol instead of the required updated configuration for external function calls, or a local memory configuration that does not contain values for all local variable in case of regular function calls. Also the choice of inputs depends on previous computation results, e.g., in case of external function calls for the previous configuration. In our model we always provide the necessary inputs to fix *any* deterministic choice in the C-IL program execution, and we use an update function instead of an updated configuration for handling external function calls. If we require inputs to contain initialization values for all local variables of *all* functions of a given program, and to contain only transition functions for external function calls that implement state transitions according to $\theta.R_{\mathbf{extern}}$, then we can exclude run-time errors due to a bad choice of inputs. The inputs for a computation can thus be chosen independently of the C-IL configuration and we will only get undefined results due to programming errors. Below we formalize the restriction on $\Sigma_{\text{C-IL}}$.

Definition 59 (C-IL Input Constrains) *We only consider input alphabets $\Sigma_{\text{C-IL}}$ that fulfill the following restrictions. For any input $(\mathcal{M}_{lvar}, \eta)$ we demand (i) that $\mathcal{M}_{lvar}$ is defined for all local variables of all internal functions of a given C-IL program π (variables have an unspecified value), and (ii) η is defined for all external functions and for all arguments its result reflects the semantics specified by $\theta.R_{\mathbf{extern}}$. For denoting that v is a local variable of function f with type t we use shorthand $lvar(v, f, t) = \exists i > \mathcal{F}_\pi^\theta(f).npar.\ \mathcal{F}_\pi^\theta(f).V[i] = (v, t)$ in:*

$$
\begin{aligned}
\Sigma_{\text{C-IL}} \equiv \{\ (\mathcal{M}_{lvar}, \eta) \ \mid\ & \textit{(i)}\ \forall f \in \mathbb{F}_{name}.\ \mathcal{F}_\pi^\theta(f).P \neq \mathbf{extern} \implies \\
& \forall (v,t) \in (\mathbb{V}, \mathbb{T}_Q).\ lvar(v,f,t) \implies \mathcal{M}_{lvar}[f](v) \in \mathbb{B}^{8 \cdot size_\theta(qt2t(t))} \\
& \textit{(ii)}\ \forall f \in \mathbb{F}_{name}.\ \mathcal{F}_\pi^\theta(f).P = \mathbf{extern} \implies \\
& \eta(f) \neq \perp \wedge \forall (X,c) \in \mathrm{dom}(\eta[f]).\ (X, c, \eta[f](X,c)) \in \theta.R_{\mathbf{extern}} \qquad \}
\end{aligned}
$$

In defining the semantics of C-IL we will use the following shorthand notation to refer to information about the topmost stack frame $top \equiv |c.s|$ in a C-IL-configuration c:

- local memory of the topmost frame: $\mathcal{M}_{\mathcal{E}\,top}(c) \equiv c.s[top].\mathcal{M}_{\mathcal{E}}$
- return destination of the topmost frame: $rds_{top}(c) \equiv c.s[top].rds$
- function name of the topmost frame: $f_{top}(c) \equiv c.s[top].f$
- location counter of the topmost frame: $loc_{top}(c) \equiv c.s[top].loc$
- function body of the topmost frame: $P_{top}(\pi, c) \equiv \pi.\mathcal{F}(f_{top}(c)).P$
- next statement to be executed: $stmt_{next}(\pi, c) \equiv P_{top}(\pi, c)[loc_{top}(c)]$

Below we define functions that perform specific updates on a C-IL configuration.

Definition 60 (Setting the Location Counter) *The function*

$$set_{loc} : conf_{\text{C-IL}} \times \mathbb{N} \rightharpoonup conf_{\text{C-IL}}$$

defined as

$$set_{loc}(c,l) \equiv c[s := (c.s)[top \mapsto (c.s[top])[loc := l]]]$$

sets the location counter of the top-most stack frame to location l.

Definition 61 (Removing the Topmost Frame) *The function*

$$drop_{frame} : conf_{\text{C-IL}} \rightharpoonup conf_{\text{C-IL}}$$

which removes the top-most stack frame from a C-IL-configuration is defined as:

$$drop_{frame}(c) \equiv c[s := c.s[1 : top)]$$

Definition 62 (Setting Return Destination) *We define the function*

$$set_{rds} : conf_{\text{C-IL}} \times (val_{\mathbf{lref}} \cup val_{\mathbf{ptr}} \cup \{\bot\}) \rightharpoonup conf_{\text{C-IL}}$$

that updates the return destination component of the top most stack frame as:

$$set_{rds}(c, v) \equiv c[s := (c.s)[top \mapsto (c.s[top])[rds := v]]]$$

Note that all of the functions defined above are only well-defined when the stack is not empty; this is why they are declared partial functions. In practice however, executing a C-IL program always requires a non-empty stack.

Definition 63 (C-IL Transition Function) *We define the transition function*

$$\delta^{\pi,\theta}_{\text{C-IL}} : conf_{\text{C-IL}} \times \Sigma \rightharpoonup conf_{\text{C-IL}}$$

by a case distinction on the given input:

- *Deterministic step, i.e., $stmt_{next}(\pi, c) \neq \mathbf{call}\ e(E)$:*

$$\delta^{\pi,\theta}_{\text{C-IL}}(c, in) = \begin{cases} inc_{loc}(c') & : stmt_{next}(\pi, c) = (e_0 = e_1) \\ set_{loc}(c, l) & : stmt_{next}(\pi, c) = \mathbf{goto}\ l \\ set_{loc}(c, l) & : stmt_{next}(\pi, c) = \mathbf{ifnot}\ e\ \mathbf{goto}\ l \wedge zero(\theta, [\![e]\!]^{\pi,\theta}_c) \\ inc_{loc}(c) & : stmt_{next}(\pi, c) = \mathbf{ifnot}\ e\ \mathbf{goto}\ l \wedge \neg zero(\theta, [\![e]\!]^{\pi,\theta}_c) \\ drop_{frame}(c) & : stmt_{next}(\pi, c) = \mathbf{return} \\ drop_{frame}(c) & : stmt_{next}(\pi, c) = \mathbf{return}\ e \wedge rds = \bot \\ write(\theta, c, rds, [\![e]\!]^{\pi,\theta}_c) & : stmt_{next}(\pi, c) = \mathbf{return}\ e \wedge rds \neq \bot \end{cases}$$

  where $c' = write(\theta, c, [\![\&(e_0)]\!]^{\pi,\theta}_c [\![e_1]\!]^{\pi,\theta}_c)$ and $rds = rds_{top}(drop_{frame}(c))$. Note that for **return** *the relevant return destination resides in the caller frame. Also, in case any of the terms used above is undefined due to run-time errors, we set $\delta^{\pi,\theta}_{\text{C-IL}}(c, in) = \bot$.*

- *Function call:*

  $\delta^{\pi,\theta}_{\text{C-IL}}(c, in)$*, where* $in.\mathcal{M}_{lvar}$ *provides initial values for all local variables of the called function, is defined if and only if all of the following hold:*

  - $stmt_{next}(\pi, c) = \textbf{call}\ e(E) \vee stmt_{next}(\pi, c) = (e_0 = \textbf{call}\ e(E))$ *— the next statement is a function call (with or without return value),*
  - $[\![e]\!]^{\pi,\theta}_c = \textbf{val}(b, \textbf{funptr}(t, T)) \wedge f = \theta.\mathcal{F}^{-1}_{adr}(b) \vee [\![e]\!]^{\pi,\theta}_c = \textbf{fun}(f, \textbf{funptr}(t, T))$ *— expression* e *evaluates to some function* f*,*
  - $|E| = \mathcal{F}^{\theta}_{\pi}(f).npar \wedge \forall i \in [1 : |E|].\ \mathcal{F}^{\theta}_{\pi}(f).V[i] = (v, t) \implies \tau_{Q^{\pi,\theta}_c}(E[i]) = t$ *— the types of all parameters passed match the declaration, and*
  - $\mathcal{F}^{\theta}_{\pi}(f).P \neq \textbf{extern}$ *— the function is not declared as external in the function table.*

  Then, we define

  $$\delta^{\pi,\theta}_{\text{C-IL}}(c, in) = c'$$

  such that

  $$c'.s = inc_{loc}(set_{rds}(c, rds)).s \circ (\mathcal{M}'_{\mathcal{E}}, \bot, f, 0)$$

  $$c'.\mathcal{M} = c.\mathcal{M}$$

  where

  $$rds = \begin{cases} [\![\&(e_0)]\!]^{\pi,\theta}_c & : \quad stmt_{next}(\pi, c) = (e_0 = \textbf{call}\ e(E)) \\ \bot & : \quad stmt_{next}(\pi, c) = \textbf{call}\ e(E) \end{cases}$$

  and

  $$\mathcal{M}'_{\mathcal{E}}(v) = \begin{cases} val2bytes_{\theta}([\![E[i]]\!]^{\pi,\theta}_c) & : \quad \exists i.\ \mathcal{F}^{\theta}_{\pi}(f).V[i] = (v, t) \wedge i < \mathcal{F}^{\theta}_{\pi}(f).npar \\ in.\mathcal{M}_{lvar}[f](v) & : \quad \exists i.\ \mathcal{F}^{\theta}_{\pi}(f).V[i] = (v, t) \wedge i \geq \mathcal{F}^{\theta}_{\pi}(f).npar \\ \text{undefined} & : \quad \text{otherwise} \end{cases}$$

- *External procedure call:*

  $\delta^{\pi,\theta}_{\text{C-IL}}(c, in)$ *is defined if and only if all of the following hold:*

  - $stmt_{next}(\pi, c) = \textbf{call}\ e(E)$ *— the next statement is a function call without return value,*
  - $[\![e]\!]^{\pi,\theta}_c = \textbf{val}(b, \textbf{funptr}(t, T)) \wedge f = \theta.\mathcal{F}^{-1}_{adr}(b) \vee [\![e]\!]^{\pi,\theta}_c = \textbf{fun}(f, \textbf{funptr}(t, T))$ *— expression* e *evaluates to some function* f*,*
  - $|E| = \mathcal{F}^{\theta}_{\pi}(f).npar \wedge \forall i \in [1 : |E|].\ \mathcal{F}^{\theta}_{\pi}(f).V[i] = (v, t) \implies \tau_{Q^{\pi,\theta}_c}(E[i]) = t$ *— the types of all parameters passed match the declaration,*
  - $in.\eta[f]((([\![E_1]\!]^{\pi,\theta}_c, \ldots, [\![E_{|E|}]\!]^{\pi,\theta}_c), c) = c'$ *— the external transition function for* f *allows a transition under given parameters* E *from* c *to* c'*,*
  - $c'.s[1 : top) = c.s[1 : top)$ *— the external procedure call does not modify any stack frames other than the topmost frame,*

- $loc_{top}(c').loc = loc_{top}(c) + 1 \wedge f_{top}(c') = f_{top}(c)$ — *the location counter of the topmost frame is incremented and the function is not changed,*
- $\mathcal{F}^{\theta}_{\pi}(f).P = \mathbf{extern}$ — *the function is declared as extern in the function table.*

Note that we restrict external function calls in such a way that they cannot be invoked with a return value. However, there is a simple way to allow an external function call to return a result: It is always possible to pass a pointer to some subvariable to which a return value from an external function call can be written.[11]

Then,

$$\delta^{\pi,\theta}_{\text{C-IL}}(c, c') = c'$$

4.3.2 C-IL Compiler Consistency

As in the MASM model we will not provide a compiler for C-IL but just state a compiler consistency relation that couples a MIPS implementation with the C-IL language level. We will later use the consistency relation to establish a simulation theorem between a C-IL *Cosmos* machine and a MIPS *Cosmos* machine, thus justifying the notion of structured parallel C, which is assumed by code verification tools like VCC.

The following theory was first documented by Andrey Shadrin [Sha12] building on previous work by W. J. Paul and others [LPP05, DPS09]. We try to stick to it as far as possible, however we take the freedom to adjust notation to fit the remainder of this thesis where we deem it useful or necessary.

Compilation and Stack Layout

We aim for a theory that is also applicable for optimizing compilers. In non-optimizing compilers, the compilation is a function mapping one C statement to a number of implementing assembly statements.[12] The compiler consistency relation between the C and the ISA level holds before and after the execution of such an assembly block. In particular the data consistency then guarantees that variables are correctly represented in memory, meaning that the memory contents agree with the values of the corresponding variables in the C semantics.

An optimizing compiler applies optimizing transformations on the compiled code of a sequence of C-IL statements, typically with the aim of reducing redundancy and the overall code execution time. Typical optimizations are, e.g., saving intermediate results of expression evaluation to reuse them for the implementation of subsequent statements, or avoiding to store frequently used data in main memory, because accesses to registers are much faster. This means however that variables are not consistent with their memory representations for most of the time. There are only a few points in a C program where the consistency relation actually holds with the optimized implementation and we call these points *compiler consistency points* or short *consistency points*.

[11] See the definition of the *cas* compiler intrinsic for an example.

[12] This is similar to the case of the MASM assembler of the previous section.

Note that in previous work [Sha12] consistency points were called *I/O points* because optimized compiler consistency was only assumed before so-called I/O steps [DPS09] which are identical to the $\mathcal{IO}$ steps in this thesis. Here we decided to disambiguate the two concepts, allowing for more optimizations by the compiler. For C-IL we assume that certain locations in a function are always consistency points. These consistency points are in particular:

- at function entry
- directly before and after function calls (including external functions)
- between two consecutive volatile variable accesses
- directly before return statements

Depending on the optimizing compiler there may be more consistency points, therefore we cannot give an exact definition of the consistency points of a given program. However, later we will introduce an uninstantiated function encoding the choice of consistency-points by the compiler and formalize above restrictions on this function.

Note that these restrictions forbid certain optimizations the compiler could apply. Sticking to [Sha12] we forbid optimizations across function calls for simplicity. The requirement of having consistency points between volatile variable accesses ensures later that the data read from or written to volatile variables is consistent with the underlying shared memory. The constraint reminds of the $\mathcal{IOIP}$ condition in the *Cosmos* model order reduction theorem, and indeed, we will later instantiate the $\mathcal{IO}$ steps of a C-IL *Cosmos* machine to be the volatile variable accesses, and the interleaving-points to be the compiler consistency points.

The C-IL compilation function now is mapping a block of C-IL statements between consistency points to blocks of assembly instructions. However as the optimizations depend on the program context we rather model the code generation as a function depending on the C-IL program, the function, and the location of the consistency point starting the block which should be compiled.

$$cpl : prog_{\text{C-IL}} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{I}^*_{\text{MIPS}}$$

This means that a C-IL function is compiled by applying the compilation function cpl subsequently on every consistency block of the function. The compiled code for the program contains the compiled code for every function positioned in a way so that jumps between functions are linked correctly.

We use the stack layout for C-IL from [Sha12] and adhere to the calling conventions CC.1 to CC.4 defined in the MASM chapter. The C-IL stack layout is depicted in Fig. 19 and is quite similar to the MASM stack layout. There are only three main differences. First, now have a place reserved in the stack frame to save the return destination, i.e., the address of the subvariable to store the return value of a function call. Secondly, in MASM the programmer of a caller function had to save and restore caller-save registers because their values are havoced by function calls. Now it is the duty of the compiler to

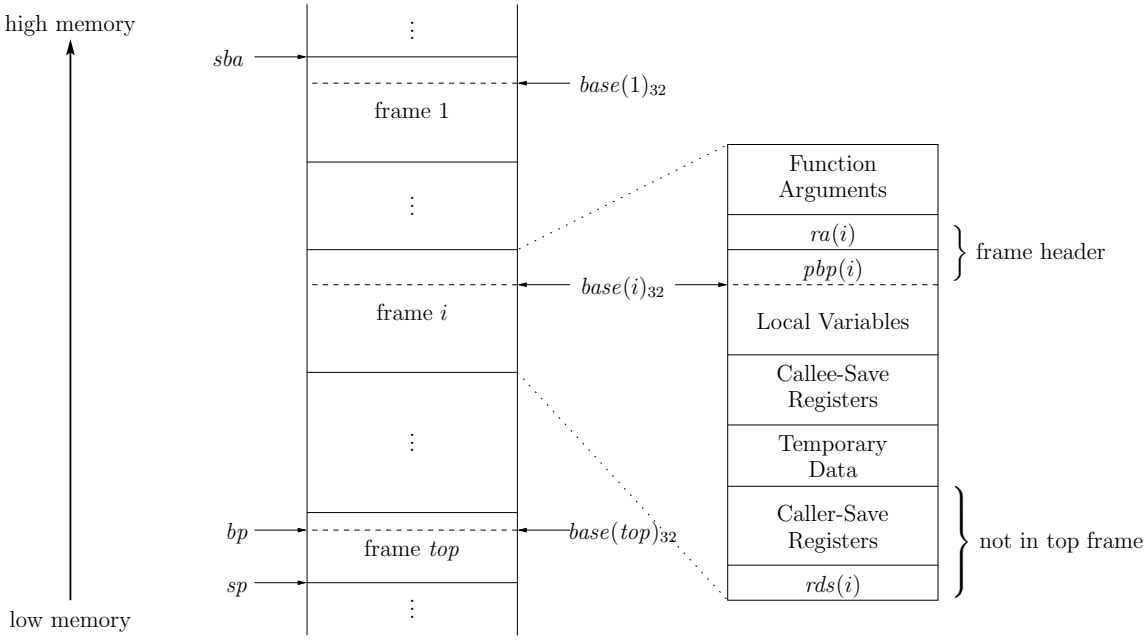

Figure 19: The C-IL stack layout.

save and restore these registers, therefore we reserve space on the stack for this purpose. Last, the local variables need to be stored on the stack as well. Generally, the C-IL stack layout exhibits the following properties.

- A C-IL frame i in memory is identified by a frame base address $base(i)_{32}$. The stack grows downwards starting at a given *stack base address* which is *not* identical with the frame base address of the first frame $base(1)_{32}$.
- The parameters for the function call are stored in the high end of the stack frame. They were stored here by the caller in reverse order. According to CC.1 the first four parameters are passed via registers i_1 to i_4, nevertheless we also reserve space for them.
- Between parameter space and the base address of a frame resides the frame header. As in MASM it contains the return address and the previous base pointer.
- The base pointer is stored in register bp and always points to the frame base address of the topmost (lowest in memory) stack frame.
- Below the frame base address we find the region of the stack frame where the local variables are saved.
- Below the local variables we the callee save registers are stored. In contrast to [Sha12] this is also done on the lowest level $i = 1$ and we assume for simplicity

that the C-IL compiler always stores all eight callee-save registers sv_1 to sv_8 in ascending order (sv_1 is at the highest memory address).

- Below that area the compiler stores temporary values in a last-in-first-out data structure. The size of the temporary area may change dynamically during program execution. This component is the similar of the *lifo* in MASM.
- The stack pointer is stored in register sp and always points on the lower end of the temporary data region of the topmost (lowest in memory) stack frame.
- In case a function is called, the compiler first stores the contents of the caller-save registers in the region directly below the temporary values.
- Next, the return destination (where the returned value of a function call should be saved) is stored by the caller. Parameters for the next frame are stored below.
- Caller-save registers, register destination and the input parameters can be seen as an extension to the *lifo*-like temporary value area, thus we are obeying calling convention CC.2. As in MASM, upon a function call from frame i the parameters become part of the next stack frame, thus they are located above the base address $base(i+1)_{32}$.
- All components of the stack are word-aligned.

Before we can formalize this notion of the stack structure in the compiler consistency relation, we need some more information about the compilation process. As in the MASM semantics we therefore introduce a C-IL compiler information data structure $info_{IL} \in InfoT_{\text{C-IL}}$, adapting the definition of Shadrin from [Sha12]. We have the following components for $info_{IL}$.

- $info_{IL}.code \in \mathbb{I}^*_{\text{MIPS}}$ — a list of MIPS instructions representing the assembled C-IL program
- $info_{IL}.cp : \mathbb{F}_{name} \times \mathbb{N} \to \mathbb{B}$ — identifies the compiler consistency points for a given function and program location.
- $info_{IL}.off : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ - A function calculating the offset in the compiled code of the first instruction which implements a C-IL statement at the specified consistency point in the given function. Note that offset 0 refers to instruction $info_{IL}.code[1]$.
- $info_{IL}.fceo : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ — the offset in the compiled code of the epilogue of a function call in a given function at a given location (see explanation below)
- $info_{IL}.lvr : \mathbb{V} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{B}^5$ — specifies, if applicable, the GPR where a given word-sized local variable of a given function is stored in a given consistency point

- $info_{IL}.lvo : \mathbb{V} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ — specifies the offset of local variables (excluding input parameters) in memory relative to the frame base for a given function and consistency point (number of bytes)
- $info_{IL}.csro : \mathbb{V} \times \mathbb{F}_{name} \times \mathbb{N} \times \mathbb{B}^5 \rightharpoonup \mathbb{N}_0$ — specifies the offset within the caller-save area where the specified register is saved by the caller during a function call in the given function and consistency point (number of bytes, counting relative to upper end with higher address)
- $info_{IL}.size_{CrS} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ — specifies the size of the caller-save region of the stack for a given caller function and location of function call (number of bytes)
- $info_{IL}.size_{tmp} : \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{N}_0$ — specifies the size of the temporary region of the stack for a given function and consistency point (number of bytes)
- $info_{IL}.cba : \mathbb{B}^{32}$ — the start address of the code region in memory
- $info_{IL}.sba : \mathbb{B}^{32}$ — the start base address
- $info_{IL}.mss : \mathbb{B}^{32}$ — the maximal size in words of the stack. We define shorthand $msp_{IL} \equiv \langle info_{IL}.sba \rangle - info_{IL}.mss + 1$ to denote the minimal allowed value for the stack pointer.

For most of the components it should be obvious why we need this information in order to define the C-IL compiler consistency relation. The only exception is maybe the *function call epilogue offset fceo*. Unlike in MASM, a function call is not completed after the return statement is executed by the callee, because the caller still has to update the return destination with the return value passed in register rv. Also the stack has to be cleared of the return destination and caller-save registers need to be restored. The code portion in the compiled code for a function call which is implementing these tasks, we call the *function epilogue*. We need to know the start of the epilogue to define the consistency relation for the return addresses.

Similar to the definition of MASM assembler consistency, we introduce notation for the frame base addresses and the distances between them. First we introduce some shorthands for the components of the i-th stack frame, implicitely depending on some C-IL configuration $c_{IL} \in conf_{\text{C-IL}}$.

$$\forall x \in \{\mathcal{M}_{\mathcal{E}}, rds, f, loc\}, i \in [1 : |c_{IL}.s|]. \quad x_i \equiv c_{IL}.s[i].x$$

Moreover let $z_i \equiv \pi.\mathcal{F}(f_i).z$ for $z \in \{V, npar\}$ denote the local variable and parameter declaration list, as well as the number of parameters for f_i. The size needed for local variables and parameters on the stack can then be computed as follows.

$$\begin{aligned} size_{par}(i) &\equiv \sum_{j=1}^{npar_i} size_{\theta}(qt2t(V_i[j].t)) \\ size_{lv}(i) &\equiv \sum_{j=npar_i+1}^{|V_i|} size_{\theta}(qt2t(V_i[j].t)) \end{aligned}$$

Here for a variable declaration $v \in \mathbb{V} \times \mathbb{T}_Q$, the notation $v.t$ refers to the type component. Then we can define the distance between base address, or between the base address of the top stack and the base pointer respectively. It is depending on c_{IL}, π, θ and $info_{IL}$. As we are not mixing notation of MASM and C-IL implementation, we can use the same names for the same concepts here without ambiguity.

$$dist(i) \equiv \begin{cases} size_{lv}(i) + 32 + info_{IL}.size_{tmp}(f_i, loc_i) & : \quad i = top \\ size_{lv}(i) + 32 + info_{IL}.size_{tmp}(f_i, loc_i) \\ + info_{IL}.size_{CrS}(f_i, loc_i) + 4 + size_{par}(f_{i+1}) + 8 & : \quad i < top \end{cases}$$

For the top frame we only store the local variables, the eight callee-save registers and temporary data in the area bounded by the addresses stored in base pointer and stack pointer. Lower frames (with lower index and higher frame base address) are storing information for the function call associated with the stack frame lying directly above (with higher index and lower frame base address). This includes the caller-save registers, the return destination. The function input parameters and the next frame header are belonging to the callee frame. For simplicity we do not make a case distinction whether a function returns a value or not. We reserve the space for the return destination in both cases. Now the frame base addresses are easily defined recursively (as natural numbers for now).

$$base(i) \equiv \begin{cases} \langle info_{IL}.sba \rangle - size_{par}(f_i) - 4 & : \quad i = 1 \\ base(i-1) - dist(i-1) & : \quad i > 1 \end{cases}$$

Similar to the MASM implementation, the frame base address of the lowest frame (with highest index in the stack) does not coincide with the stack base address because we need space above the frame base to store the input parameters and the frame header. Again we define shorthands for return address, previous base pointer, and also the return destination, depending on some ISA configuration $h \in \mathbb{H}_{\text{MIPS}}$.

$$\begin{aligned} \forall i \in [1 : |c_{IL}.s|].\ ra(i) &\equiv h.m_4((base(i) + 4)_{32}) \\ \forall i \in [1 : |c_{IL}.s|).\ rds(i) &\equiv h.m_4((base(i) + 8 + size_{par}(i))_{32}) \\ \forall i \in [1 : |c_{IL}.s|].\ pbp(i) &\equiv h.m_4(base(i)_{32}) \end{aligned}$$

Compiler Consistency Points

We want to restrict the function $info_{IL}.cp$ such that compiler consistency points occur at least at the required positions in the code as described above. However, to this end we first need a predicate which detects whether a given C-IL statement s contains a volatile variable access. This is the case if expressions in s reference variables that have quantified type (q, t) with $\mathbf{volatile} \in q$. Unfortunately, the type evaluation function $\tau_{Q_c^{\pi,\theta}}$ depends on a given C-IL configuration c. Nevertheless, the configuration is only used in the type evaluation of variables, where the type is looked up either from the

global variable declaration, or from the local variable declaration in the function table for the function in the topmost stack frame of c. Thus the type of an expression can be determined without a C-IL configuration if the function is known, in whose context the expression is evaluated. Indeed we could define such a function $\tau_{Q_f}^{\pi,\theta}$ in the same way as $\tau_{Q_c}^{\pi,\theta}$ is defined in [Sch13b], by simply replacing any reference to $f_{top}(c)$ by f. Then we directly get for all $e \in \mathbb{E}$:

$$\tau_{Q_c}^{\pi,\theta}(e) \;=\; \tau_{Q_{f_{top}(c)}}^{\pi,\theta}(e)$$

Using this modified type evaluation function we can define a predicate which detects accesses to volatile variales in expressions that occur in the code of a C-IL function f.

Definition 64 (Expression Contains Volatile Variables) *Given a C-IL expression e that is evaluated in the context of a function f of C-IL program π wrt. environments parameters θ. Then e contains an access to a volatile variable in case the following predicate is fulfilled.*

$$vol_f^{\pi,\theta}(e) \equiv$$
$$\begin{cases} \mathbf{volatile} \in q & : \quad e \in \mathbb{V} \wedge \tau_{Q_f}^{\pi,\theta}(e) = (q,t) \\ vol_f^{\pi,\theta}(e') & : \quad (\exists \ominus \in \mathbb{O}_1.\ e = \ominus e') \vee e = \&(*(e')) \\ & \quad \vee\ e = \mathbf{sizeof}(e') \vee e = (t)e' \\ vol_f^{\pi,\theta}(e') \vee vol_f^{\pi,\theta}(e'') & : \quad \exists \oplus \in \mathbb{O}_2.\ e = e' \oplus e'' \\ vol_f^{\pi,\theta}(e') \vee vol_f^{\pi,\theta}(e'') \vee vol_f^{\pi,\theta}(e''') & : \quad e = (e'\ ?\ e'' : e''') \\ \mathbf{volatile} \in q \vee vol_f^{\pi,\theta}(e') & : \quad e = *(e')\ \wedge \tau_{Q_f}^{\pi,\theta}(e') = (q', \mathbf{ptr}(q,t)) \\ & \quad \vee\ e = (e').f' \wedge \tau_{Q_f}^{\pi,\theta}(e) = (q,t) \\ 0 & : \quad \text{otherwise} \end{cases}$$

Note that the evaluation of constants, function names, addresses of variables, and type casts do not require volatile variable accesses in general. For most of the other cases the above definition meets what one would expect intuitively. Nevertheless there are two cases worth mentioning.

First, as pointer dereferencing may involve two accesses to memory, there are also two possibilities for a volatile access. Both the pointer as well as the referenced subvariable might be volatile. Considering field accesses we also have several possibilities. On the one hand the field itself might be declared volatile. On the other hand the contained struct may be volatile, or the evaluation of the reference to that containing struct variable may involve volatile accesses, respectively.

Observe also, that due to possible pointer arithmetic, the definition only covers all volatile variable accesses in an expression, if it is typed correctly. In case arithmetic expressions that target volatile variables are not type-cast using the **volatile** quantifier, we cannot detect these accesses statically. However we will later define the ownership policy in a way that such accesses are treated as safety violations.
Below we can can now define a similar predicate to statically detect volatile variable accesses in C-IL statements of a given program.

Definition 65 (Statement Accesses Volatile Variables) *Given a C-IL statement s that is executed in the context of a function f of C-IL program π wrt. environments parameters θ. Then f accesses a volatile variable in case the following predicate is fulfilled.*

$$vol_f^{\pi,\theta}(s) \equiv \begin{cases} vol_f^{\pi,\theta}(e) & : \quad s \in \{\textbf{ifnez } e \textbf{ goto } l, \textbf{return } e\} \\ vol_f^{\pi,\theta}(e) \vee vol_f^{\pi,\theta}(e') & : \quad s \equiv (e = e') \\ \bigvee_{e'=e \vee e' \in E} vol_f^{\pi,\theta}(e') & : \quad s \equiv \textbf{call } e(E) \\ \bigvee_{e'' \in \{e,e'\} \vee e'' \in E} vol_f^{\pi,\theta}(e'') & : \quad s \equiv (e' = \textbf{call } e(E)) \\ 0 & : \quad \text{otherwise} \end{cases}$$

Note that according to this definition a C-IL statement may contain more than one volatile variable access. Nevertheless, as updates to volatile variables must be implemented as atomic operations, we will later formulate the restriction that there may be at most one volatile variable access per C-IL statement. Also, for simplicity, we will only consider volatile variable accesses in assignments. Now we can, however, define the required consistency points for a given C-IL program.

Definition 66 (Required C-IL Compiler Consistency Points) *Given compiler information $info_{IL}$ for a C-IL program π and environment parameter θ, the following predicate holds, iff there are compiler consistency points (i) at the entry of every function, (ii) before and after function calls, (iii) between any two consecutive volatile variable accesses, and (iv) before return statements. Let $s_{f,i} = \pi.\mathcal{F}(f).P[i]$, $call(s) = \exists e, e', E.\ s \in \{\textbf{call } e(E), (e'' = \textbf{call } e(E))\}$, and $ret(s) = \exists e.\ s \in \{\textbf{return}, \textbf{return } e\}$ in:*

$$\begin{aligned} req\mathcal{CP}(\pi, \theta, info_{IL}) \equiv\ & \forall f \in \mathbb{F}_{name}, i \in \mathbb{N}.\ \pi.\mathcal{F}(f).P \neq \textbf{extern} \wedge i \leq |\pi.\mathcal{F}(f).P| \implies \\ & \textit{(i)} \quad info_{IL}.cp(f, 1) \\ & \textit{(ii)} \quad call(s_{f,i}) \implies info_{IL}.cp(f, i) \wedge info_{IL}.cp(f, i+1) \\ & \textit{(iii)} \quad vol_f^{\pi,\theta}(s_{f,i}) \wedge (\exists j < i.\ vol_f^{\pi,\theta}(s_{f,j})) \implies \exists k \in (j : i].\ info_{IL}.cp(f, k) \\ & \textit{(iv)} \quad ret(s_{f,i}) \implies info_{IL}.cp(f, i) \end{aligned}$$

In what follows we assume that a C-IL compiler obeys these rules in the selection of consistency points for a given C-IL program.

Compiler Consistency Relation

Now we will define the compiler consistency relation that links a C-IL computation to its implementation on the MIPS ISA level. We want to relate a C-IL configuration c_{IL} to an ISA state h that implements the program π using the environment parameters θ and compiler information $info_{IL}$. Formally we thus define a simulation relation

$$consis_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, h)$$

stating the consistency between these entities. The relation is supposed to hold only in compiler consistency points, which are identified by a function name and a location

according to the $info_{IL}.cp$ predicate. We define the following predicate which holds iff c_{IL} is currently in a consistency point.

$$cp(c_{IL}, info_{IL}) \stackrel{def}{\equiv} info_{IL}.cp(f_{top}(c_{IL}), loc_{top}(c_{IL}))$$

The compiler consistency relation is split in two sub-relations covering control and data consistency. The first part talks about control-flow and is thus concerned with the program counter and the return address. Let the following function compute the start address of the compiled code for the C-IL statements starting from consistency point loc in function f.

$$adr(info_{IL}, f, loc) \stackrel{def}{\equiv} info_{IL}.cba +_{32} (4 \cdot info_{IL}.off(f, loc))_{32}$$

Definition 67 (C-IL Control Consistency) *We define control consistency sub-relation for C-IL $consis^{control}_{\text{C-IL}}$, which states that (i) the program counter of the MIPS machine must point to the start of the compiled code for the current statement in the C-IL machine which is at a compiler consistency point. In addition (ii) the return address of any stack frame is pointing to the beginning of the function call epilogue for the function call statement in the previous frame (with lower index).*

$$\begin{aligned} &consis^{control}_{\text{C-IL}}(c_{IL}, info_{IL}, h) \equiv \\ &\quad \textit{(i)} \quad cp(c_{IL}, info_{IL}) \wedge h.pc = adr(info_{IL}, f_{top}, loc_{top}) \\ &\quad \textit{(ii)} \quad \forall i \in (1, |c_{IL}.s|].\ ra(i) = info_{IL}.cba +_{32} (4 \cdot info_{IL}.fceo(f_{i-1}, loc_{i-1} - 1))_{32} \end{aligned}$$

According to the C-IL semantics, the current location of a caller frame already points to the statement after the function call (which is a consistency point). To obtain the location of the function call we therefore have to subtract 1 from that location. When control returns to the caller frame, on the ISA level first the function call epilogue is executed before the consistency point is reached.

Data consistency is split into several parts covering registers, the global memory, local variables, the code region as well as the stack structure. The register consistency relation covers only the stack and base pointers.

Definition 68 (C-IL Register Consistency) *The C-IL register consistency relation demands, that (i) the base pointer points to the base address of the top frame, while (ii) the stack pointer points to the top-most element of the temporary values (growing downwards) in the top frame.*

$$\begin{aligned} consis^{regs}_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, h) \stackrel{def}{\equiv} \quad &\textit{(i)} \quad h.gpr(bp) = bin_{32}(base(top)) \\ &\textit{(ii)} \quad h.gpr(sp) = bin_{32}(base(top) - dist(top)) \end{aligned}$$

In the code consistency relation we also need to couple π with the compiled code.

Definition 69 (C-IL Code Consistency) *For C-IL code consistency we require that (i) the compiler consistency points were selected by the compiler according to our requirements, (ii)the*

compiled code in the compiler information is actually corresponding to the C-IL program, and that (iii) the compiled code is converted to binary format and resides in a contiguous region in the memory of the MIPS machine starting at the code base address.

$$
\begin{aligned}
&\mathit{consis}^{\mathit{code}}_{\text{C-IL}}(c_{IL}, \pi, \theta, \mathit{info}_{IL}, h) \stackrel{\mathit{def}}{\equiv} \\
&\quad \textit{(i)} \quad \mathit{reqCP}(\pi, \theta, \mathit{info}_{IL}) \\
&\quad \textit{(ii)} \quad \forall f \in \mathrm{dom}(\pi.\mathcal{F}), l.\ \ \mathit{info}_{IL}.\mathit{cp}(f, l) \implies \\
&\qquad\qquad \forall i \in [1 : |\mathit{cpl}(\pi, f, l)|].\ \mathit{info}_{IL}.\mathit{code}[\mathit{info}_{IL}.\mathit{off}(p, l) + i] = \mathit{cpl}(\pi, f, l)[i] \\
&\quad \textit{(iii)} \quad \forall j \in [0 : |\mathit{info}_{IL}.\mathit{code}|). \\
&\qquad\qquad \mathit{info}_{IL}.\mathit{code}[j + 1] = \mathit{decode}(h.m_4(\mathit{info}_{IL}.\mathit{cba} +_{32} (4 \cdot j)_{32}))
\end{aligned}
$$

Again, the latter property forbids self-modifying code in case info_{IL} is fixed for a simulation between the MIPS and C-IL machines. We redefine the shorthands CR and StR to represent the code region, or the region where the C-IL stack is allocated respectively.

$$
\begin{aligned}
CR &\equiv [\langle \mathit{info}_{IL}.\mathit{cba} \rangle : \langle \mathit{info}_{IL}.\mathit{cba} \rangle + 4 \cdot |\mathit{info}_{IL}.\mathit{code}|) \\
StR &\equiv [\mathit{msp}_{IL} : \langle \mathit{info}_{IL}.\mathit{sba} \rangle]
\end{aligned}
$$

Now we demand memory consistency for all addresses but the code region and the stack region, because these addresses may not be accessed directly in C-IL programs.

$$
\mathit{consis}^{\mathit{mem}}_{\text{C-IL}}(c_{IL}, \mathit{info}_{IL}, h) \stackrel{\mathit{def}}{\equiv} \forall \mathit{ad} \in \mathbb{B}^{32}.\ \langle \mathit{ad} \rangle \notin CR \cup StR \implies h.m(\mathit{ad}) = c_{IL}.\mathcal{M}(\mathit{ad})
$$

Note that this definition includes the consistency for global variables since they are always allocated in the global memory $c.\mathcal{M}$. The allocated address for a given global variable is determined by a global variable allocation function $\theta.\mathit{alloc}_{gv} : \mathbb{V} \rightharpoonup \mathbb{B}^{32}$. We did not introduce it in the C-IL semantics because it is only relevant for the definition of expression evaluation, which we excluded from our presentation.

In contrast to global variables, local variables are allocated on the stack using offsets from $\mathit{info}_{IL}.\mathit{lvo}$. Moreover top frame local variables and parameters may be kept in registers according to the compiler information $\mathit{info}_{IL}.\mathit{lvr}$. In [Sha12] the local variable consistency relation did not talk about the frames below the top frame (*caller frames*), however, such a compiler consistency relation is not inductive in the sense that it cannot be used in an inductive compiler correctness proof. When treating **return** instructions one cannot establish the local variable consistency for the new top frame without knowing where the values of the local variables of that frame were stored before returning.

In fact for the local variables and parameters of caller stack frames there are three possibilities depending on where they are expected to be stored upon return from the called function. If they are supposed to be allocated on the stack upon function return, then we demand that they already reside in their dedicated stack location during the execution of the callee. If they are to be allocated in caller-save registers, we require the caller to store them in its caller-save area during the function call. Similarly we demand

the callee to store them in the callee-save area if we expect their value to reside in callee-save registers after returning from the function call. Below we give a correct definition of the C-IL local variable consistency relation.

Definition 70 (C-IL Local Variable Consistency) *Compiler consistency relation* $consis^{lv}_{\text{C-IL}}$ *couples the values of local variables (including parameters) of stack frames with the MIPS ISA implementation. Let*

$$\begin{aligned}
(v_{i,j}, t_{i,j}) &\equiv V_i[j] \\
r_{i,j} &\equiv info_{IL}.lvr(v_{i,j}, f_i, loc_i) \\
lva_{i,j} &\equiv bin_{32}(base(i) - 4 \cdot info_{IL}.lvo(v_{i,j}, f_i, loc_i)) \\
para_{i,j} &\equiv bin_{32}\left(base(i) + 8 + \sum_{k=1}^{j-1} size_\theta(qt2t(t_{i,k}))\right) \\
crsbase_i &\equiv base(i) - 4 \cdot (|V_i| - npar_i) - 32 - info_{IL}.size_{tmp}(f_i, loc_i) - 4 \\
crsa_{i,j} &\equiv bin_{32}\left(csrbase_i - info_{IL}(v_{i,j}, f_i, loc_i, r_{i,j})\right) \\
csa_{i,j} &\equiv bin_{32}\left(base(i) - 4 \cdot (|V_{i+1}| - npar_{i+1}) - 4 \cdot \epsilon\{k \in \mathbb{N}_8 \mid r_{i,j} = sv_k\}\right)
\end{aligned}$$

where $v_{i,j}$ *is the* j*-th local variable in frame* i *with type* $t_{i,j}$*, that is allocated on the stack if* $r_{i,j}$ *is undefined. Then it is stored at local variable address* $lva_{i,j}$ *or parameter address* $para_{i,j}$*. In the other case that* $r_{i,j}$ *is defined, variables of the top frame are stored in the corresponding registers. Variables of other stack frames that are allocated in registers are stored either in the caller-save area starting from (upper) base address* $crsbase_i$ *at address* $crsa_{i,j}$*, or in the callee-save area of the callee frame at address* $csa_{i,j}$*. Formally, with* $CS = \{sv_1, \ldots, sv_8\}$*:*

$$\begin{aligned}
consis^{lv}_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, h) &\stackrel{def}{\equiv} \forall i \in \mathbb{N}_{top}, j \in \mathbb{N}_{|V_i|}. \\
\implies \mathcal{M}_{\mathcal{E}i}(v_{i,j}) &= \begin{cases}
h.c.gpr(r_{i,j}) & : \quad r_i \neq \bot \wedge i = top \\
h.m_4(csa_{i,j}) & : \quad r_{i,j} \in CS \wedge i < top \\
h.m_4(crsa_{i,j}) & : \quad r_{i,j} \in \mathbb{B}^5 \setminus CS \wedge i < top \\
h.m_{size_\theta(qt2t(t_{i,j}))}(lva_{i,j}) & : \quad r_{i,j} = \bot \wedge j > npar_i \\
h.m_{size_\theta(qt2t(t_{i,j}))}(para_{i,j}) & : \quad \text{otherwise}
\end{cases}
\end{aligned}$$

Note above that wrt. local variable consistency, all caller frames rest in the consistency point that is placed immediately after the function call they initiated, because of the C-IL semantics for the location counter. In case a function is called, loc is already advanced behind the function call statement. Thus the allocation of variables for caller frames is determined by the compiler information of the the consistency point that will be reached after returning from the called function. This setting, together with the definition of $consis^{lv}_{\text{C-IL}}$, places several restrictions on the implementation of function calls.

First, when a local variable is supposed to be allocated on the stack upon returning from the function, it has to be stored in the right position throughout the complete function call. This means that the compiler has to store it on the stack before jumping to the callee function. Similarly, when variables are supposed to reside in callee-save or

caller-save registers upon return, the compiler needs to put them into the right callee-save register, or store them in the caller-save area, respectively, before the jump.

Moreover we restricted the optimizing compiler by demanding that it always saves all eight callee-save registers in the callee-save area. A lazier implementation might just keep them in the registers if they are not modified. In case of further function calls their values would be preserved by the calling convention. Such a setting would lead to a much more complex situation where local variables of caller frames on the bottom of the stack may be stored in much higher stack frames or even the registers of the top frame. In order to keep the definitions simple, we did not allow such optimizations here. The consistency relation for the remaining stack components is stated below.

Definition 71 (CIL Stack Consistency) *The C-IL stack component is implemented correctly in memory, if in every stack frame except the lowest one (i) the previous base pointer field contains the address of the base of the previous frame (with higher index), and if (ii) the return destination points to the correct address, according to the rds component of the C-IL function frame, in case it is defined. Let $alv = bin_{32}(base(j) - info_{IL}.lvo(v, f_j, loc_j) + o)$ in:*

$$consis^{stack}_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, h) \equiv \forall i \in [1 : |c_{IL}.s|).$$

$$\textit{(i)} \quad pbp(i+1) = base(i)$$

$$\textit{(ii)} \quad rds(i) \neq \bot \implies rds(i) = \begin{cases} a & : \quad rds(i) = \mathbf{val}(a,t) \in val_{\mathbf{ptr}} \\ alv & : \quad rds(i) = \mathbf{lref}((v,o),j,t) \in val_{\mathbf{lref}} \end{cases}$$

Now we can collect all sub-relations and define the overall compiler consistency relation between C-IL and MIPS configurations.

Definition 72 (C-IL Compiler Consistency Relation) *The C-IL consistency relation comprises the consistency between MIPS and C-IL machine wrt. (i) program counter and return addresses, (ii) the code region, (iii) stack and base pointer registers, (iv) the global memory region, (v) the local variables and parameters, as well as (vi) return destinations and the chain of previous base pointers.*

$$consis_{\text{C-IL}}(c_{IL}, \pi, \theta, info_{IL}, h) \equiv$$

$$\begin{array}{llll} \textit{(i)} & consis^{control}_{IL}(c_{IL}, info_{IL}, h) & \textit{(iv)} & consis^{mem}_{IL}(c_{IL}, info_{IL}, h) \\ \textit{(ii)} & consis^{regs}_{IL}(c_{IL}, \pi, \theta, info_{IL}, h) & \textit{(v)} & consis^{lv}_{IL}(c_{IL}, \pi, \theta, info_{IL}, h) \\ \textit{(iii)} & consis^{code}_{IL}(c_{IL}, \pi, \theta, info_{IL}, h) & \textit{(vi)} & consis^{stack}_{IL}(c_{IL}, \pi, \theta, info_{IL}, h) \end{array}$$

Simulation Theorem

For stating the simulation theorem between C-IL and MIPS ISA we again have to introduce well-formedness conditions on the C-IL and MIPS configurations and computations. In order to enable a simulation, like in MASM, we require that C-IL programs do not exceed the maximal stack size and do not directly access the stack or code regions of memory. We adapt the stack overflow predicate to the C-IL case.

$$stackovf(c_{IL}, \pi, \theta, info_{IL}) \stackrel{def}{\equiv} (base(1) - dist(1)) < msp_{IL}$$

In order to be able to detect bad memory accesses to the stack or code region we must define the memory footprint of C-IL expressions and statements. First we introduce a function to compute the memory region occupied by referenced global subvariables.

Definition 73 (Footprint Function for Global Subvariables) *Let $a \in \mathbb{B}^{32}$ be an address that a pointer variable points to and $t \in \{\mathbf{ptr}(t'), \mathbf{array}(t', n)\}$ the type of that pointer. Then the memory footprint of the referenced subvariable is computed by the following function.*

$$fp_\theta(a,t) \equiv \begin{cases} [\langle a \rangle : \langle a \rangle + size_\theta(t')) & : \quad /isarray(t') \\ \emptyset & : \quad \text{otherwise} \end{cases}$$

Arrays cannot be accessed as a whole, we only read their elements using pointer arithmetic. Therefore we define the memory footprint of array variables to be empty.

Definition 74 (Global Memory Footprint of C-IL Expressions) *Function*

$$A^{\cdot,\cdot}_{\cdot}(\cdot) : conf_{\text{C-IL}} \times prog_{\text{C-IL}} \times params_{\text{C-IL}} \times \mathbb{E} \rightharpoonup 2^{[0:2^{32})}$$

computes the set of global memory addresses that are accessed when evaluating a given C-IL expression e wrt. some C-IL configuration c, program π, and environment parameters θ as follows. Let $sv(e) \equiv e \in \mathbb{V} \vee \exists e' \in \mathbb{E}, f \in \mathbb{F}.\ e = (e').f$ (i.e., e is a subvariable) in:

$$A^{\pi,\theta}_c(e) \equiv \begin{cases} \emptyset & : \quad e \in val \cup \mathbb{F}_{name} \vee \exists t \in \mathbb{T}_Q.\ e = \mathbf{sizeof}(t) \\ & \quad \vee\ sv(e) \wedge [\![\&(e)]\!]^{\pi,\theta}_c \in val_{\mathbf{lref}} \\ & \quad \vee\ \exists e' \in \mathbb{E}.\ e \in \{\&(e'), \mathbf{sizeof}(e')\} \wedge sv(e') \\ fp_\theta(a,t) & : \quad sv(e) \wedge [\![\&(e)]\!]^{\pi,\theta}_c = \mathbf{val}(a,t) \in val_{\mathbf{ptr}} \\ A^{\pi,\theta}_c(e') & : \quad \exists \ominus \in \mathbb{O}_1, t \in \mathbb{T}_Q.\ e \in \{\ominus e', e = (t)e', e = \&(*(e'))\} \\ & \quad \vee\ e = *(e') \wedge [\![e']\!]^{\pi,\theta}_c \in val_{\mathbf{lref}}\ \vee e = \mathbf{sizeof}(*(e')) \\ A^{\pi,\theta}_c(x) \cup A^{\pi,\theta}_c(e') & : \quad \exists e'' \in \mathbb{E} \wedge e = (x\ ?\ e' : e'') \wedge zero(\theta, [\![x]\!]^{\pi,\theta}_c) \\ A^{\pi,\theta}_c(x) \cup A^{\pi,\theta}_c(e'') & : \quad \exists e' \in \mathbb{E} \wedge e = (x\ ?\ e' : e'') \wedge /zero(\theta, [\![x]\!]^{\pi,\theta}_c) \\ A^{\pi,\theta}_c(e') \cup A^{\pi,\theta}_c(e'') & : \quad \exists \oplus \in \mathbb{O}_2.\ e = e' \oplus e'' \\ A^{\pi,\theta}_c(e') \cup fp_\theta(a,t) & : \quad e = *(e') \wedge [\![e']\!]^{\pi,\theta}_c = \mathbf{val}(a,t) \in val_{\mathbf{ptr}} \\ \bot & : \quad \text{otherwise} \end{cases}$$

The definition is straight-forward for most of the cases. Unlike global subvariables, C-IL values and function names are not associated with any memory address. The same holds for local subvariables. Looking up addresses and type sizes does not touch memory either. In order to dereference a pointer to a global memory location one must evaluate the address to be read, but also read the memory region referenced by that typed pointer. We need another predicate to detect whether some expression encodes a reference to the *cas* intrinsic function. Let the type signature of the *cas* intrinsic be denoted by $t_{cas} = \mathbf{funptr}(\mathbf{void}, \mathbf{ptr}(\mathbf{i32}) \circ \mathbf{i32} \circ \mathbf{i32} \circ \mathbf{ptr}(\mathbf{i32}))$. Then we define:

$$cas^{\pi,\theta}_c(e) \quad \stackrel{def}{\equiv} \quad \exists b \in \mathbb{B}^{32}.\ [\![e]\!]^{\pi,\theta}_c = \mathbf{val}(b, t_{cas}) \wedge \theta.\mathcal{F}^{-1}_{adr}(b) = cas \vee [\![e]\!]^{\pi,\theta}_c = \mathbf{fun}(cas, t_{cas})$$

Now the memory footprint of a C-IL statement is easily defined using the expression footprint notation.

Definition 75 (Memory Footprint of C-IL statements) *We overload the definition of function $A_c^{\pi,\theta}$ from above to cover also C-IL statements $s \in \mathbb{I}_{\text{C-IL}}$. Let $A_E = \bigcup_{e \in E} A_c^{\pi,\theta}(e)$ for $E \in \mathbb{E}^*$ as well as $A_{cas} = A_c^{\pi,\theta}(*a) \cup A_c^{\pi,\theta}(u) \cup A_c^{\pi,\theta}(v) \cup A_c^{\pi,\theta}(*r)$ in:*

$$A_c^{\pi,\theta}(s) \equiv \begin{cases} \emptyset & : \quad s = \textbf{return} \vee \exists l \in \mathbb{N}.\ s = \textbf{goto}\ l \\ A_c^{\pi,\theta}(e) & : \quad \exists l \in \mathbb{N}.\ s = \textbf{ifnez}\ e\ \textbf{goto}\ l \\ & \quad \vee s = \textbf{return}\ e \wedge \mathit{rds}(\mathit{top}-1) \in \mathit{val}_{\textbf{lref}} \\ A_c^{\pi,\theta}(e) \cup \mathit{fp}_\theta(a,t) & : \quad s = \textbf{return}\ e \wedge \mathit{rds}(\mathit{top}-1) = \textbf{val}(a,t) \in \mathit{val}_{\textbf{ptr}} \\ A_c^{\pi,\theta}(e) \cup A_c^{\pi,\theta}(e') & : \quad s = (e = e') \\ A_c^{\pi,\theta}(e) \cup A_E & : \quad s = \textbf{call}\ e(E) \wedge / \mathit{cas}_c^{\pi,\theta}(e) \\ A_c^{\pi,\theta}(e) \cup A_{cas} & : \quad s = \textbf{call}\ e(a,u,v,r) \wedge \mathit{cas}_c^{\pi,\theta}(e) \\ A_c^{\pi,\theta}(e) \cup A_c^{\pi,\theta}(e') \cup A_E & : \quad s = (e' = \textbf{call}\ e(E)) \\ \bot & : \quad \text{otherwise} \end{cases}$$

For most statements the footprint of the C-IL statements is only depending on the expressions they are containing. Only the return statement which returns a value writes additional memory cells. In the special case of cas we know from its semantics that also the memory locations referenced by inputs a and r are accessed. With the above definition all C-IL software conditions for simulation can be summed up below.

Definition 76 (C-IL Software Conditions) *A C-IL program can be implemented if all reachable configurations obey the software conditions denoted by the following predicate. Given a C-IL configuration c_{IL}, programm π, environment parameters θ, and assembler information info_{IL}, then the next step according to input $\mathit{in} \in \Sigma_{\text{C-IL}}$ may (i) not produce a run-time error, (ii) not result in a stack overflow, and (iii) not explicitely access the stack or code region. Additionally, in (ii) we demand the minimal stack pointer value to be positive, and that (iv) the code region fits into memory and is disjoint from the stack region.*

$$\begin{aligned} \mathit{sc}_{\text{C-IL}}(c_{IL}, \mathit{in}, \pi, \theta, \mathit{info}_{IL}) \equiv \quad & \textit{(i)} \quad \delta^{\pi,\theta}_{\text{C-IL}}(c_{IL}, \mathit{in}) \neq \bot \\ & \textit{(ii)} \quad / \mathit{stackovf}(c_{IL}, \pi, \theta, \mathit{info}_{IL}) \wedge \mathit{msp}_{IL} \geq 0 \\ & \textit{(iii)} \quad A^{\pi,\theta}_{c_{IL}}(\mathit{stmt}_{\mathit{next}}(\pi, c_{IL})) \cap (\mathit{CR} \cup \mathit{StR}) = \emptyset \\ & \textit{(iv)} \quad \mathit{CR} \subseteq [0 : 2^{32}) \wedge \mathit{CR} \cap \mathit{StR} = \emptyset \end{aligned}$$

Note that these restrictions imply that accessed global variables are not allocated in the stack or code region by the compiler. Also, by (i) the software conditions exclude common programming errors like out-of-bounds array accesses, or dereferencing dangling pointers to local variables.

Another software condition one could think of is to limit the number of global variables so that all fit in global memory. However this is already covered here because

of two facts. First, in C-IL semantics there is an explicit allocation function $\theta.alloc_{gv}$ for global variables which determines their addresses in global memory. Secondly, the absence of run-time errors ensures that every global variable that is ever accessed is actually allocated. Thus we cannot have too many global variables in a program that is fulfilling the software conditions stated above.

Concerning the well-formedness of MIPS configurations and well-behaving of MIPS computations, we have the same conditions as for MASM, ensuring that no external or internal interrupts are triggered and that instructions are fetched from the code region.

$$\begin{aligned} suit^{\text{C-IL}}_{\text{MIPS}}(eev) &\stackrel{def}{\equiv} /eev[0] \\ wb^{\text{C-IL}}_{\text{MIPS}}(h, eev) &\stackrel{def}{\equiv} /jisr(h.c, I(h), eev) \wedge [\langle h.c.pc\rangle : \langle h.c.pc\rangle + 3] \subseteq CR \\ wf^{\text{C-IL}}_{\text{MIPS}}(h) &\stackrel{def}{\equiv} h.spr(sr)[\text{dev}] = 0 \end{aligned}$$

Now we can state the sequential simulation theorem for C-IL which is mostly similar to the MASM theorem. The main difference is that the n ISA steps do not simulate one but many C-IL steps, because we are dealing with an optimizing compiler. Note that, similar to the MASM simulation theorem, we only cover one simulation step which can be used in an inductive simulation proof between C-IL and MIPS computations.

Theorem 3 (Sequential C-IL Simulation Theorem) *Given an initial C-IL configuration $c_{IL0} \in conf_{\text{C-IL}}$ that is (i) well-formed, a MIPS configuration $h_0 \in \mathbb{H}_{\text{MIPS}}$ that is (ii) well-formed for C-IL simulation and (iii) consistent to c_{IL0} wrt. some $\pi \in Prog_{\text{C-IL}}$, $\theta \in params_{\text{C-IL}}$, and $info_{IL} \in InfoT_{\text{C-IL}}$. If (iv) every C-IL state reachable from c_{IL0} is well-formed and fulfilling the C-IL software conditions,*

$$\begin{aligned} &\forall c_{IL0}, \pi, \theta, h_0, info_{IL}. \\ &\qquad \textit{(i)} \quad wf_{\text{C-IL}}(c_{IL0}, \pi, \theta) \\ &\qquad \textit{(ii)} \quad wf^{\text{C-IL}}_{\text{MIPS}}(h_0) \\ &\qquad \textit{(iii)} \quad consis_{\text{C-IL}}(c_{IL0}, \pi, \theta, info_{IL}, h_0) \\ &\qquad \textit{(iv)} \quad \forall c'_{IL} \in Prog_{\text{C-IL}}, n \in \mathbb{N}_0, in \in (\Sigma_{\text{C-IL}})^{n+1}.\ c_{IL0} \longrightarrow^n_{\delta_{\text{C-IL}}, in} c'_{IL} \implies \\ &\qquad\qquad wf_{\text{C-IL}}(c'_{IL}, \pi, \theta) \wedge sc_{\text{C-IL}}(c'_{IL}, in_{n+1}, \pi, \theta, info_{IL}) \end{aligned}$$

then there exists an ISA computation that (i) starts in h_0 and preserves well-formedness. Moreover (ii) there is a well-formed C-IL state c'_{IL} obtained by taking steps from c_{IL}. The implementing ISA computation is (iii) well-behaved and (iv) leading into a state consistent with c'_{IL}.

$$\begin{aligned} \implies \quad &\exists n \in \mathbb{N}, h \in \mathbb{H}^{n+1}_{\text{MIPS}}, eev \in (\mathbb{B}^{256})^n, c'_{IL} \in conf_{\text{C-IL}}, m \in \mathbb{N}_0, in \in (\Sigma_{\text{C-IL}})^m. \\ &\qquad \textit{(i)} \quad h_1 = h_0 \wedge h_1 \longrightarrow^n_{\delta_{\text{MIPS}}, eev} h_{n+1} \wedge wf^{\text{C-IL}}_{\text{MIPS}}(h_{n+1}) \\ &\qquad \textit{(ii)} \quad c_{IL0} \longrightarrow^m_{\delta^{\pi,\theta}_{\text{C-IL}}, in} c'_{IL} \wedge wf_{\text{C-IL}}(c'_{IL}, \pi, \theta) \\ &\qquad \textit{(iii)} \quad \forall i \in \mathbb{N}_n.\ wb^{\text{C-IL}}_{\text{MIPS}}(h^i, eev^i) \\ &\qquad \textit{(iv)} \quad consis_{\text{C-IL}}(c'_{IL}, \pi, \theta, info_{IL}, h_{n+1}) \end{aligned}$$

Note that hypothesis (iv) could not be discharged if we used the definition of C-IL from [Sch13b], because there run-time errors can occur due to a bad choice of inputs for a computation, as explained before.
For proving the theorem one needs to know the code generation function of a given optimizing C-IL compiler and prove the correctness of the generated code for statements between consistency points. Then using code consistency one argues that only the correct generated code is executed, eventually leading into another consistency point.

Again the theorem allows us to couple uninterrupted sequential MIPS ISA computations with a corresponding C-IL computation. Any uninterrupted ISA computation of a big enough length, that is running out of a consistency point, contains the simulating ISA computation from the theorem as a prefix because without external inputs any ISA computation is only depending on the initial configuration. Thus by induction on the number of consistency points passed one can repeat this argument and find the C-IL computation that is simulated by the original ISA computation. We will formalize this notion for concurrently executing program threads in the next chapter. Before that we conclude by presenting the instantiation of our *Cosmos* model with C-IL.

4.3.3 Concurrent C-IL

In what follows we define a *Cosmos* machine $S^n_{\text{C-IL}} \in \mathbb{S}$ which contains n C-IL computation units working on a shared global memory. This concurrent C-IL model will be justified later by a simulation proof between $S^n_{\text{C-IL}}$ and S^n_{MIPS} using the sequential simulation theorem defined above (cf. Section 5.6.2). Like in the concurrent MASM model all C-IL units share the same program and environment parameters, but they are running on different stacks, since each unit can be in a different program state. Hence we have disjoint stack regions in memory with different stack base addresses but the same length. The instantiation is thus based on the parameters $\pi \in Prog_{\text{C-IL}}$, $\theta \in params_{\text{C-IL}}$, $info^p_{IL} \in InfoT_{\text{C-IL}}$ for $p \in \mathbb{N}_n$. The compiler information is equal for all units except for the stack base address.

$$\forall q, r \in \mathbb{N}_n, c. \quad q \neq r \wedge c \neq sba \implies info^q_{IL}.c = info^r_{IL}.c$$

Thus we can refer to a common compiler information data structure $info_{IL}$ which is consistent with all $info^p_{IL}$ wrt. all components but sba. We redefine the shorthands for stack and code regions below, adapting them to the C-IL setting.

$$\begin{aligned} CR &\stackrel{def}{\equiv} [\langle info_{IL}.cba\rangle : \langle info_{IL}.cba\rangle + 4 \cdot |info_{IL}.code|) \\ StR_p &\stackrel{def}{\equiv} (\langle info^p_{IL}.sba\rangle - info_{IL}.msp_{IL} : \langle info^p_{IL}.sba\rangle] \end{aligned}$$

Then required disjointness of stack frames in memory is denoted by:

$$\forall q, r \in \mathbb{N}_n.\ q \neq r \implies StR_q \cap StR_r = \emptyset$$

Before, we already noted that the software conditions on C-IL enforce that no global variables are allocated in the stack or code memory region. However this is only guaranteed for global variables that are actually accessed in the program. For instantiation

of our *Cosmos* model we need to make the requirement explicit. Let $StR \equiv \bigcup_{p=1}^{n} StR_p$ be the complete stack region and let

$$A_{gv}^{\theta}(v,t) \stackrel{def}{\equiv} [\langle \theta.alloc_{gv}(v) \rangle : \langle \theta.alloc_{gv}(v) \rangle + size_{\theta}(qt2t(t)))$$

be the address range allocated for some global variable $v \in \mathbb{V}$ of qualified type $t \in \mathbb{T}_Q$. Then we require:

$$\forall (v,t) \in \pi.V_G.\ A_{gv}^{\theta}(v,t) \cap (CR \cup StR) = \emptyset$$

We now define the components of $S^n_{\text{C-IL}}$ one by one.

- $S^n_{\text{C-IL}}.\mathcal{A} = \{a \in \mathbb{B}^{32} \mid \langle a \rangle \notin CR \cup StR\}$ and $S^n_{\text{C-IL}}.\mathcal{V} = \mathbb{B}^8$ — as for MASM we obtain the memory for the C-IL system by cutting out the forbidden address ranges for the stack and code regions from the underlying MIPS memory.
- $S^n_{\text{C-IL}}.\mathcal{R} = \{a \in \mathbb{B}^{32} \mid \exists (v,(q,t)) \in \pi.V_G.\ \langle a \rangle \in A_{gv}^{\theta}(v,(q,t)) \wedge \mathbf{const} \in q\}$ — as constants are supposed to never change their values, we should forbid writing them via the ownership policy by including them in the read-only set. This way, ownership safety guarantees the absence of writes to constant global variables, that cannot be detected by static checks of the compiler. For simplicity we exclude here constant subvariables of global variables that are not constant. We could easily cover them as well, after introducing the environment parameter which determines the offsets of fields in composite-type variables. Nevertheless, note that the ownership policy cannot exclude writes to local or dynamically allocated constant variables, because on the one hand local variables are not allocated in the global memory and the ownership policy only governs global memory accesses. On the other hand, the set of read-only addresses is fixed in our ownership model, thus we cannot add new constant variables to $\mathcal{R}$.
- $S^n_{\text{C-IL}}.nu = n$ — We have n C-IL computation units.
- $S^n_{\text{C-IL}}.\mathcal{U} = frame^*_{\text{C-IL}} \cup \{\bot\}$ — Each C-IL computation unit is either in a run-time error state $\bot$, or it consists of a C-IL stack component upon which it bases its local computations.
- $S^n_{\text{C-IL}}.\mathcal{E} = \Sigma_{\text{C-IL}}$ — The external input alphabet for the C-IL transition function is also suitable for the C-IL *Cosmos* machine
- $S^n_{\text{C-IL}}.reads$ — We need to specify the explicit read accesses to global memory that are associated with the next C-IL step of a given unit. We introduce the reads-set function for C-IL statements

  $$R^{\cdot,\cdot}_{\cdot}(\cdot) : conf_{\text{C-IL}} \times prog_{\text{C-IL}} \times params_{\text{C-IL}} \times \mathbb{I}_{\text{C-IL}} \rightharpoonup 2^{[0:2^{32})}$$

  which defined similarly to the memory footprint of C-IL statements but excludes write accesses. Note that the global memory is only updated by variable assignments, return statements with a return value and the *cas* primitive. For all other statements we can use the memory footprint function defined earlier.

In case of a Compare-and-Swap intrinsic function call of $cas(a, u, v, r)$, also the memory location referenced by a is read but the target of r is only written.

For the return statements we simply exclude the memory location referenced by the rds component of the previous stack frame to determine the corresponding reads-set.

For assignments we need to exclude the written memory cells which are specified in the left hand side of the assignment. However we cannot simply exclude the left hand side expression from the computation of the reads-set, since there might be read accesses necessary in order to evaluate it. Therefore we perform a case distinction on e in $s = (e = e')$:

1. e is a plain variable identifier — Then no additional global memory cells need to be read in order to obtain the variable's address in memory.
2. e is dereferencing a pointer expression — Then there might be further memory reads necessary in order to evaluate the pointer expression. However the referenced memory location is not added to the reads-set explicitely. It still might occur in the reads-set though if it contributes to the evaluation of the pointer expression.
3. e contains either a redundant $\&$-$*$-combination or references a field of a subvariable — In the first case expression evaluation simply discards the redundant $\&$-$*$-combination. In the latter case one first has to evaluate the referenced subvariable before the address of the field can be computed. In both cases the expression evaluation step does not require memory accesses, hence we continue to compute the reads-set recursively with the inner subexpression which must specify a subvariable.

Using $A_{cas} = A_c^{\pi,\theta}(*a) \cup A_c^{\pi,\theta}(u) \cup A_c^{\pi,\theta}(v) \cup A_c^{\pi,\theta}(r)$ we define these ideas formally:

$$R_c^{\pi,\theta}(s) \stackrel{def}{\equiv} \begin{cases} A_c^{\pi,\theta}(e') & : \exists e \in \mathbb{V}.\ s = (e = e') \\ A_c^{\pi,\theta}(e') \cup A_c^{\pi,\theta}(e'') & : s = (*(e'') = e') \\ R_c^{\pi,\theta}((e = e')) & : \exists f \in \mathbb{F}.\ s \in \{(\&(*(e)) = e'), ((e).f = e')\} \\ A_c^{\pi,\theta}(e) & : s = \textbf{return}\ e \\ A_c^{\pi,\theta}(e) \cup A_{cas} & : s = \textbf{call}\ e(a, u, v, r) \wedge cas_c^{\pi,\theta}(e) \\ A_c^{\pi,\theta}(s) & : \text{otherwise} \end{cases}$$

Remember that C-IL configurations $c_{IL} = (\mathcal{M}, s)$ consist of a memory $\mathcal{M} : \mathbb{B}^{32} \to \mathbb{B}^8$ and a stack $s \in frame^*_{\text{C-IL}}$, thus a pair $(\lceil m \rceil, u)$ consisting of a completed partial *Cosmos* machine memory $m : \mathcal{A} \rightharpoonup \mathcal{V}$ and a unit configuration $u \in \mathcal{U}$ represents a proper C-IL configuration. Now the $reads$ function of the *Cosmos* machine can easily be instantiated.

$$S^n_{\text{C-IL}}.reads(u, m, in) = R^{\pi,\theta}_{(\lceil m \rceil, u)}(stmt_{next}(\pi, (\lceil m \rceil, u)))$$

- $S^n_{\text{C-IL}}.\delta$ — We simply use the C-IL transition function $\delta^{\pi,\theta}_{\text{C-IL}}$ in the instantiation of the transistion function for the C-IL computation units. Again, we need to fill the partial memory that is given to the $S^n_{\text{C-IL}}.\delta$ as an input with dummy values, so that we can apply $\delta^{\pi,\theta}_{\text{C-IL}}$ on it. Moreover, we need to define the writes-set for a given C-IL statement s because the output memory of the transition function needs to be restricted to this set. As noted above, only assigments, cas, and certain return statements may modify the global memory. Let $X = ([\![a]\!]^{\pi,\theta}_{c_{IL}}, [\![u]\!]^{\pi,\theta}_{c_{IL}}, [\![v]\!]^{\pi,\theta}_{c_{IL}}, [\![r]\!]^{\pi,\theta}_{c_{IL}})$ in the predicate

$$cas^{\pi,\theta}_{c_{IL}}(s, a, u, v, r, \rho, in) \stackrel{def}{\equiv} s = \textbf{call}\ e(a, u, v, r) \wedge cas^{\pi,\theta}_{c_{IL}}(e) \wedge (X, c_{IL}, in.\eta[cas](X, c_{IL})) \in \rho$$

which denotes that statement s is a call to the cas intrinsic function with the specified input parameters, and that the external function call has an effect according to transition relation $\rho \in \theta.R_{\textbf{extern}}$. Thus we define the writes-set for a given C-IL statement.

$$W^{\pi,\theta}_{c_{IL}}(s, in) \stackrel{def}{\equiv} \begin{cases} fp_\theta(a,t) & : \exists e, e' \in \mathbb{E}.\ s = (e = e') \wedge [\![\&(e)]\!]^{\pi,\theta}_{c_{IL}} = \textbf{val}(a,t) \in val_{\textbf{ptr}} \\ & \quad \vee s = \textbf{return}\ e \wedge rds_{top-1} = \textbf{val}(a,t) \in val_{\textbf{ptr}} \\ fp_\theta(x,t) \cup fp_\theta(y,t) & : \exists a, u, v, r \in \mathbb{E}.\ cas^{\pi,\theta}_{c_{IL}}(s, a, u, v, r, \rho^{swap}_{cas}, in) \\ & \quad \wedge [\![a]\!]^{\pi,\theta}_{c_{IL}} = \textbf{val}(x,t) \wedge [\![r]\!]^{\pi,\theta}_{c_{IL}} = \textbf{val}(y,t) \wedge t = \textbf{ptr}(\textbf{i32}) \\ fp_\theta(x, \textbf{ptr}(\textbf{i32})) & : \exists a, u, v, r \in \mathbb{E}.\ cas^{\pi,\theta}_{c_{IL}}(s, a, u, v, r, \rho^{swap}_{cas}, in) \\ & \quad \wedge [\![a]\!]^{\pi,\theta}_{c_{IL}} = \textbf{val}(x, \textbf{ptr}(\textbf{i32})) \wedge [\![r]\!]^{\pi,\theta}_{c_{IL}} \in val_{\textbf{lref}} \\ fp_\theta(y, \textbf{ptr}(\textbf{i32})) & : \exists a, u, v, r \in \mathbb{E}.\ cas^{\pi,\theta}_{c_{IL}}(s, a, u, v, r, \rho^{fail}_{cas}, in) \\ & \quad \wedge [\![r]\!]^{\pi,\theta}_{c_{IL}} = \textbf{val}(y, \textbf{ptr}(\textbf{i32})) \\ \emptyset & : \text{otherwise} \end{cases}$$

In the first case either execution is returning from a function call with a return value that is written to the memory cells specified by the return destination of the caller function frame, or we have an assignment to a memory location. The remaining cases deal with the various outcomes of a Compare-and-Swap intrinsic function call. If the comparison was successful the targeted shared memory location is written. Also we must distinguish whether the value read for comparison is returned to a local or global variable. Only in the latter case the variable update contributes to the writes-set.

Now we can define the transition function for C-IL computation units with the following case distinction. Let $W = W^{\pi,\theta}_{(\lceil m \rceil, u)}(stmt_{next}(\pi, (\lceil m \rceil, u)), in)$ in:

$$S^n_{\text{C-IL}}.\delta(u, m, in) = \begin{cases} (\mathcal{M}'|_W, s') & : \quad \delta^{\pi,\theta}_{\text{C-IL}}((\lceil m \rceil, u), in) = (\mathcal{M}', s') \\ \bot & : \quad c = \bot \vee \delta^{\pi,\theta}_{\text{C-IL}}((\lceil m \rceil, u), in) = \bot \end{cases}$$

If $c = \perp$ or $\delta_{\text{C-IL}}$ returns $\perp$, so does $S^n_{\text{C-IL}}.\delta(u, m, in)$. Note that in contrast to the C-IL semantics, we do not update the complete memory for external function calls, because doing so would break the ownership memory access policy. Instead we only update the relevant memory portions according to the semantics of the particular external function, i.e., of cas in this case. This approach is sound because we have defined the external transition function input η in such a way, that it implements the semantics specified by $\theta.R_{\mathbf{extern}}$.

- $S^n_{\text{C-IL}}.\mathcal{IO}$ — There are basically two kinds of $\mathcal{IO}$ steps in C-IL. As in MASM we consider the use of the Compare-and-Swap mechanism as an $\mathcal{IO}$ step. Moreover we include accesses to volatile subvariables. As we have seen above C-IL statements may depend on a lot of variables and memory locations, therefore, in order to avoid spurious accesses to shared memory, we restrict the way we may safely access volatile variables in C-IL. We set up the following rules which will be enforced by the ownership discipline when defining the $\mathcal{IO}$ predicate accordingly.
  - Volatile variables may only be accessed in assignment statements or by the intrinsic function cas.
  - Per assignment there may be only one access to a volatile variable.
  - The right hand side of assignments with a volatile read, is either a volatile variable identifier, or it is dereferencing a pointer expression which is *either* volatile *or* pointing to a volatile variable.
  - The left hand side of assignments with a volatile write has the same form.

  Note that this excludes references to volatile variables in function calls, **return** statements, and **goto** conditions. We do not support these cases here for simplicity. Above, we already introduced the predicate $vol^{\pi,\theta}_f$ which scans expressions of a C-IL function f recursively for volatile variable accesses. We derive a similar predicate for evaluating expressions in the top frame of configuration c_{IL}.

$$vol^{\pi,\theta}_{c_{IL}}(e) \stackrel{def}{\equiv} vol^{\pi,\theta}_{f_{top}(c_{IL})}(e)$$

  Now we can define the $\mathcal{IO}$ predicate, formalizing the rules stated above. Let $no2vol^{\pi,\theta}_{c_{IL}}(e'') \equiv \exists q, q', t.\ \tau_{Q\,c_{IL}}^{\pi,\theta}(e'') = (q', \mathbf{ptr}(q,t)) \wedge \mathbf{volatile} \notin q \cap q'$ in:

$$\begin{aligned}
&S^n_{\text{C-IL}}.\mathcal{IO}(u, m, in) = 1 \quad \Longleftrightarrow \\
&\quad \exists e, e', e'' \in \mathbb{E}, E \in \mathbb{E}^*. \\
&\qquad stmt_{next}(\pi, (\lceil m \rceil, u)) = \mathbf{call}\ e(E) \wedge cas^{\pi,\theta}_{(\lceil m \rceil, u)}(e) \\
&\qquad \vee\ stmt_{next}(\pi, (\lceil m \rceil, u)) \in \{(e = e'), (e' = e)\} \\
&\qquad\quad \wedge / vol^{\pi,\theta}_{c_{IL}}(e) \wedge vol^{\pi,\theta}_{c_{IL}}(e') \wedge (e' \in \mathbb{V} \vee (e' = *(e'')) \wedge no2vol(e''))
\end{aligned}$$

  Note, that any access to shared memory which does not obey the rules above will not be considered an $\mathcal{IO}$ step and thus be unsafe according to the ownership memory access policy.

- $S^n_{\text{C-IL}}.\mathcal{IP}$ — Again we could choose the interleaving-points to be $\mathcal{IO}$ points to allow for coarse scheduling and an easier verification of concurrent C-IL code. However, later we want to show a simulation between the concurrent MIPS and the concurrent C-IL model, so we have to choose consistency points as interleaving-points such that in the *Cosmos* model we interleave blocks of code that are executed by different C-IL units and each block starts in a consistency point.

  $$S^n_{\text{C-IL}}.\mathcal{IP}(u, m, in) = info_{IL}.cp(f_{top}(\lceil m \rceil, u), loc_{top}(f_{top}(\lceil m \rceil, u)))$$

  With this definition of $\mathcal{IO}$ steps and interleaving-points we can make sure by the verification of ownership safety, that shared variables are only accessed at a few designated points in the program, which are chosen by the programmer. This allows on the one hand for the efficient verification of concurrent C-IL programs, on the other hand it enables us to justify the concurrent C-IL model, using our order reduction theorem.

  In order to do so we would first need to determine the aforementioned set A_{io} of the underlying MIPS *Cosmos* machine (cf. Sect. 4.1.4). Since all assignments contain only one access to a volatile variable the compiler can ensure the same for the compiled code. Since there is a consistency point before every assignment that includes a volatile variable, we can determine the address of the memory instruction implementing the access with the help of $info_{IL}.cba$, $info_{IL}.off$, and the code compilation function. We collect all these instruction addresses in A_{io}. The set A_{cp}, which contains the addresses of all consistency points in the machine code, can easily be defined using $info_{IL}.cba$, $info_{IL}.off$, and $info_{IL}.cp$. For a formal description of the simulation between $S^n_{\text{C-IL}}$ and S^n_{MIPS} see Sect. 5.6.2.

Thus we have instantiated our *Cosmos* machine with the C-IL semantics obtaining a concurrent C-IL model. However we still need to discharge $insta_r(S^n_{\text{C-IL}})$ which demands the following property of our $reads$ function instantiation. We have to prove that if the memories of two C-IL machines $(\lceil m \rceil, u)$ and $(\lceil m' \rceil, u)$ agree on reads-set $R = S^n_{\text{C-IL}}.reads(u, m, in)$ of the first machine, then both machines are reading the same addresses in the next step.

$$m|_R = m'|_R \implies S^n_{\text{C-IL}}.reads(u, m', in) = R$$

In contrast to MASM, the reads-set of a C-IL step depends heavily on memory because of expression evaluation which constitutes in fact a series of memory read accesses that are depending on each other. Thus we can see that the statement above is a suitable correctness condition on the reads-set instantiation. If it did not hold, i.e., if both machines would read different addresses although they have the same stack and their memories agree on the reads-set of one machine, then this can only be explained by a dependency of the reads-set on some memory location that is not covered by the read-set definition. This however means that this definition is incorrect since it does not cover all memory addresses that the next step depends upon.

We finish the instantiation chapter by a proof of the instantiation constraint on the C-IL reads-set. To this end we first prove several lemmas about the C-IL memory footprint and reads-set notation.

Lemma 25 (C-IL Expression Memory Footprint Correctness) *Given are a C-IL program π, environment parameters θ and two C-IL configurations c, c' that agree on their stack, i.e., $c.s = c'.s$, and a C-IL expression e. If the memories of both machines agree on the memory footprint of e wrt. configuration c, then e is evaluated to the same value in both configurations. Let $A = \{a \in \mathbb{B}^{32} \mid \langle a \rangle \in A_c^{\pi,\theta}(e)\}$ in:*

$$c.\mathcal{M}|_A = c'.\mathcal{M}|_A \implies [\![e]\!]_c^{\pi,\theta} = [\![e]\!]_{c'}^{\pi,\theta}$$

PROOF SKETCH: Since we have not defined expression evaluation, we cannot give a formal proof here, however we can provide a proof sketch. The proof is conducted by a structural induction on C-IL expressions. There we have the following base cases.

- If e is a proper C-IL value then expression evaluation returns just e again in both configurations.
- If e is a function name, a function pointer value is returned after a function table look-up. As in this evaluation memory is not concerned, again the same values are computed in c and c'.
- If $e = \mathit{sizeof}(t)$ for some qualified type t, similarly only a type table lookup is necessary for the evaluation and we get the same result.
- If e is taking the address of some variable name v, i.e., $e = \&(v)$, then the following subcases are base.
  - If v is a local variable in the top frame then it is so in both configurations since the stacks are equal in c and c'. A local reference is returned according to the type declaration of the top function. Again memory is not concerned here so the same value is returned.
  - The same holds for global variables but in addition the address of the global variable has to be computed using the global variable allocation function.
- If e is a variable name we first evaluate $\&(e)$ as above. If the resulting value is a pointer to some primitive value or a storable (i.e., not symbolic) function pointer value, then the global memory footprint $A_c^{\pi,\theta}(e)$ of the variable is part of A according to the definition of $A_c^{\pi,\theta}$. Then the same value is read from memory when evaluating e in both configurations. This is also the case when e identifies a local variable, because we have the same stacks in c and c'. For array variables a pointer to the start of the array is returned independent of the state of memory.

In the induction step we assume that the claim holds for all subexpressions of e. Note that by the recursive definition of the global memory footprint function (cf. Def. 74), the memory footprints of these sub expressions are also contained in A and the memories in c and c' agree on them. We have the following cases for e:

- For unary or binary mathematical expressions we apply the induction hypothesis (IH) on the operands and apply the corresponding operator semantics function in order to obtain the same result of expression evaluation in both configurations. The same principle holds for type casts and the case $e = \mathbf{sizeof}(e')$.
- If e is the ternary *if-then-else* operator we first argue by induction hypothesis that the *if*-condition is consistently evaluated in c and c'. Consequently the same decision is made whether to evaluate the *then*-expression or the *else*-expression. Also the footprint then contains only the addresses relevant for the chosen subexpression. Applying IH again on this subexpression yields our claim.
- In case $e = *(e')$ is dereferencing a pointer, then we know that the same pointer value is dereferenced in c and c' according to IH on e'. The remaining proof is similar to the last base case.
- If $e = \&(*(e'))$ then the $\&$-$*$-combination is simply redundant and skipped. By IH e' is evaluated consistently in both configurations c and c'.
- If $e = \&((e').f)$ is the address of a field, then we first evaluate $\&(e')$ which results in the same pointer value according to IH. Then we simply add the appropriate offset from the struct type declaration, which does not depend on memory. If $e = (e').f$ is identifying a field subvariable we use the same approach to get a pointer to that field. We then dereference the pointer as above getting the same results in both configurations because the memory footprint of the subvariable is contained in A by definition.
- In all other cases the expression evaluation is undefined and it is so in both configurations. Thus we know that the expression evaluation of any C-IL expression is depending only on the stack and the expression's memory footprint. □

We can use this result in proving the following lemma.

Lemma 26 (C-IL Expression Memory Footprint Agreement) *Given are a C-IL program π, environment parameters θ and two C-IL configurations c, c' that agree on their stack, i.e., $c.s = c'.s$, and a C-IL expression e. If the memories of both machines agree on the memory footprint of e wrt. configuration c, then the footprints of e agree in both configurations. Let $A = \{a \in \mathbb{B}^{32} \mid \langle a \rangle \in A_c^{\pi,\theta}(e)\}$ in:*

$$c.\mathcal{M}|_A = c'.\mathcal{M}|_A \implies A_c^{\pi,\theta}(e) = A_{c'}^{\pi,\theta}(e)$$

PROOF: Again we prove the lemma by structural induction on C-IL expressions and fill in the definition of their memory footprints. We have the following base cases:

- $e \in val \cup \mathbb{F} \vee e = \mathbf{sizeof}(t)$ — Here e has an empty footprint in both configurations.

$$A_c^{\pi,\theta}(e) = \emptyset = A_{c'}^{\pi,\theta}(e)$$

- $sv(e) \wedge [\![\&(e)]\!]_c^{\pi,\theta} = \mathbf{val}(a,t) \in val_{\mathbf{ptr}}$ — By Lemma 25 we have $[\![\&(e)]\!]_c^{\pi,\theta} = [\![\&(e)]\!]_{c'}^{\pi,\theta} = \mathbf{val}(a,t)$, thus we have the same footprint, independent of memory.

$$A_c^{\pi,\theta}(e) = fp_\theta(a,t) = A_{c'}^{\pi,\theta}(e)$$

- $sv(e) \wedge [\![\&(e)]\!]_c^{\pi,\theta} \in val_{\mathbf{lref}} \vee \exists e' \in \mathbb{E}.\ e \in \{\&(e'), \mathbf{sizeof}(e')\} \wedge sv(e')$ — In the first case we argue as above, but the expression identifies a local variable which is stored on the stack and not in global memory. The evaluation of addresses and type sizes for subvariables does not depend on memory, hence the footprint is empty as well in both configurations according to the definition of $A_c^{\pi,\theta}(e)$.

$$A_c^{\pi,\theta}(e) = \emptyset = A_{c'}^{\pi,\theta}(e)$$

In the induction step we assume that the claim holds for all subexpressions of e.

- $\exists \ominus \in \mathbb{O}_1, t \in \mathbb{T}_Q.\ e \in \{\ominus e', e = (t)e', e = \&(*(e'))\} \vee e = *(e') \wedge [\![e']\!]_c^{\pi,\theta} \in val_{\mathbf{lref}} \vee e = \mathbf{sizeof}(*(e'))$ — Lemma 25 yields $[\![e']\!]_{c'}^{\pi,\theta} \in val_{\mathbf{lref}}$, hence we have the same cases in both machines. Induction hypothesis and the definition of $A_c^{\pi,\theta}$ gives us:

$$A_c^{\pi,\theta}(e) = A_c^{\pi,\theta}(e') \overset{\text{IH}}{=} A_{c'}^{\pi,\theta}(e') = A_{c'}^{\pi,\theta}(e)$$

- $e = (x\ ?\ e' : e'') \wedge zero(\theta, [\![x]\!]_c^{\pi,\theta})$ — By Lemma 25 we obtain $[\![x]\!]_c^{\pi,\theta} = [\![x]\!]_{c'}^{\pi,\theta}$. Then also $zero(\theta, [\![x]\!]_{c'}^{\pi,\theta})$ holds. Consequently in both c and c' we compute the footprint for e including expression e' and leaving out e''. By definition and IH:

$$A_c^{\pi,\theta}(e) = A_c^{\pi,\theta}(x) \cup A_c^{\pi,\theta}(e') \overset{\text{IH}}{=} A_{c'}^{\pi,\theta}(x) \cup A_{c'}^{\pi,\theta}(e') = A_{c'}^{\pi,\theta}(e)$$

- $e = (x\ ?\ e' : e'') \wedge /zero(\theta, [\![x]\!]_c^{\pi,\theta})$ — This case is symmetric to the one above.
- $e = e' \oplus e''$ — Once more we use the memory footprint definition and IH.

$$A_c^{\pi,\theta}(e) = A_c^{\pi,\theta}(e') \cup A_c^{\pi,\theta}(e'') \overset{\text{IH}}{=} A_{c'}^{\pi,\theta}(e') \cup A_{c'}^{\pi,\theta}(e'') = A_{c'}^{\pi,\theta}(e)$$

- $e = *(e') \wedge [\![e']\!]_c^{\pi,\theta} = \mathbf{val}(a,t) \in val_{\mathbf{ptr}}$ — From Lemma 25 we get $[\![e']\!]_{c'}^{\pi,\theta} = \mathbf{val}(a,t)$, consequently we are dereferencing the same pointer in c' and we conclude:

$$A_c^{\pi,\theta}(e) = A_c^{\pi,\theta}(e') \cup fp_\theta(a,t) \overset{\text{IH}}{=} A_{c'}^{\pi,\theta}(e') \cup fp_\theta(a,t) = A_{c'}^{\pi,\theta}(e)$$

- In the remaining faulty cases $A_c^{\pi,\theta}(e)$ and $A_{c'}^{\pi,\theta}(e)$ are both undefined, hence we have shown for all cases of e that the computation of a C-IL expression's memory footprint is only depending on the addresses contained in the footprint. □

We show the same property for memory footprints of C-IL statements.

Lemma 27 (C-IL Statement Memory Footprint Agreement) *Given are a C-IL program π, environment parameters θ and two C-IL configurations c, c' that agree on their stack, i.e., $c.s = c'.s$, and a C-IL statement s. If the memories of both machines agree on the memory footprint of s wrt. configuration c, then the footprints of s agree in both configurations. Let $A = \{a \in \mathbb{B}^{32} \mid \langle a \rangle \in A_c^{\pi,\theta}(e)\}$ in:*

$$c.\mathcal{M}|_A = c'.\mathcal{M}|_A \implies A_c^{\pi,\theta}(s) = A_{c'}^{\pi,\theta}(s)$$

PROOF: by case distinction on statement s.

- $s \in \{\mathbf{return}, \mathbf{goto}\ l\}$ — These statements have an empty footprint by definition.

$$A_c^{\pi,\theta}(s) = \emptyset = A_{c'}^{\pi,\theta}(s)$$

- $s = \mathbf{return}\ e \wedge c.s[top{-}1].rds = \mathbf{val}(a,t) \in val_{\mathbf{ptr}}$ — From Lemma 26 we get $A_c^{\pi,\theta}(e) = A_{c'}^{\pi,\theta}(e)$ therefore:

$$A_c^{\pi,\theta}(s) = A_c^{\pi,\theta}(e) \overset{\text{L26}}{=} A_{c'}^{\pi,\theta}(e) = A_{c'}^{\pi,\theta}(s)$$

- $s = \mathbf{ifnez}\ e\ \mathbf{goto}\ l \vee s = \mathbf{return}\ e \wedge c.s[top{-}1].rds \in val_{lref}$ — From Lemma 26 we again get $A_c^{\pi,\theta}(e) = A_{c'}^{\pi,\theta}(e)$, then definition yields:

$$A_c^{\pi,\theta}(s) = A_c^{\pi,\theta}(e) \cup fp_\theta(a,t) \overset{\text{L26}}{=} A_{c'}^{\pi,\theta}(e) \cup fp_\theta(a,t) = A_{c'}^{\pi,\theta}(s)$$

- $s = (e = e)$ — By Lemma 26, expressions e and e' have the same footprints in c and c'. Using the defintion of memory footprints we conclude:

$$A_c^{\pi,\theta}(s) = A_c^{\pi,\theta}(e) \cup A_c^{\pi,\theta}(e') \overset{\text{L26}}{=} A_{c'}^{\pi,\theta}(e) \cup A_{c'}^{\pi,\theta}(e') = A_{c'}^{\pi,\theta}(s)$$

- $s = \mathbf{call}\ e(E) \wedge /cas_c^{\pi,\theta}(e)$ — The predicate $cas_c^{\pi,\theta}$ is defined as follows for type signature t_{cas} introduced earlier.

$$\exists b \in \mathbb{B}^{32}.\ [\![e]\!]_c^{\pi,\theta} = \mathbf{val}(b, t_{cas}) \wedge \theta.\mathcal{F}_{adr}^{-1}(b) = cas \vee [\![e]\!]_c^{\pi,\theta} = \mathbf{fun}(cas, t_{cas})$$

  From Lemma 25 we get $[\![e]\!]_c^{\pi,\theta} = [\![e]\!]_{c'}^{\pi,\theta}$. Thus we also have $cas_c^{\pi,\theta}(e) = cas_{c'}^{\pi,\theta}(e)$ and $/cas_{c'}^{\pi,\theta}(e)$ holds. For all input parameters $i \in E$ we get $A_c^{\pi,\theta}(i) = A_{c'}^{\pi,\theta}(i)$. Hence we conclude:

$$A_c^{\pi,\theta}(s) = A_c^{\pi,\theta}(e) \cup \bigcup_{i \in E} A_c^{\pi,\theta}(i) \overset{\text{L26}}{=} A_{c'}^{\pi,\theta}(e) \cup \bigcup_{i \in E} A_{c'}^{\pi,\theta}(i) = A_{c'}^{\pi,\theta}(s)$$

  The other cases for function calls are proven similarly.

- The footprints are undefined in all other cases both in c and c', hence we have proven our claim for all s. □

Now we can show the required property for the reads-set of C-IL statements.

Lemma 28 (C-IL Reads-Set Agreement) *Given are a C-IL program π, environment parameters θ and two C-IL configurations c, c' that agree on their stack, i.e., $c.s = c'.s$, and a C-IL statement s. If the memories of both machines agree on reads-set of s wrt. configuration c, the reads-sets of s agree in both configurations. Let $R = \{a \in \mathbb{B}^{32} \mid \langle a \rangle \in R_c^{\pi,\theta}(e)\}$ in:*

$$c.\mathcal{M}|_R = c'.\mathcal{M}|_R \implies R_c^{\pi,\theta}(s) = R_{c'}^{\pi,\theta}(s)$$

PROOF: by case distinction on s:

- $s \in \{(e' = e), \mathbf{return}\ e\}$ — Since by definition $A_c^{\pi,\theta}(e) \subseteq R$ we have $c.\mathcal{M}|_{A_c^{\pi,\theta}(e)} = c.\mathcal{M}'|_{A_c^{\pi,\theta}(e)}$. Thus we can apply Lemma 26 and by the definition of $R_c^{\pi,\theta}$ we have:

$$R_c^{\pi,\theta}(s) = A_c^{\pi,\theta}(e) \overset{\text{L26}}{=} A_{c'}^{\pi,\theta}(e) = R_{c'}^{\pi,\theta}(s)$$

- $s = (*(e'') = e')$ — Again we use Lemma 26 and the definition of $R_c^{\pi,\theta}$:

$$R_c^{\pi,\theta}(s) = A_c^{\pi,\theta}(e') \cup A_c^{\pi,\theta}(e'') \overset{\text{L26}}{=} A_{c'}^{\pi,\theta}(e') \cup A_{c'}^{\pi,\theta}(e'') = R_{c'}^{\pi,\theta}(s)$$

- $s \in \{(\&(*(e)) = e'), ((e).f = e')\}$ — We recursively apply the definition of $R_c^{\pi,\theta}$ removing all field selectors or redundant $\&$-$*$-pairs from e until we obtain an assignment $(x = e')$, where x is either a variable identifier or it is dereferencing a pointer. Both cases where proven above.

- $s = \mathbf{call}\ e(a, u, v, r) \wedge cas_c^{\pi,\theta}(e)$ — As we have seen in the previous proof, the equality of $A_c^{\pi,\theta}(e)$ and $A_c^{\pi,\theta}(e')$ that we get from Lemma 26 implies that $cas_{c'}^{\pi,\theta}(e)$ holds. Also the input parameters of the cas function have the same memory footprints in both machines, therefore we conclude:

$$\begin{aligned} R_c^{\pi,\theta}(s) &= A_c^{\pi,\theta}(e) \cup A_c^{\pi,\theta}(*a) \cup A_c^{\pi,\theta}(u) \cup A_c^{\pi,\theta}(v) \cup A_c^{\pi,\theta}(r) \\ &\overset{\text{L26}}{=} A_{c'}^{\pi,\theta}(e) \cup A_{c'}^{\pi,\theta}(*a) \cup A_{c'}^{\pi,\theta}(u) \cup A_{c'}^{\pi,\theta}(v) \cup A_{c'}^{\pi,\theta}(r) \\ &= R_{c'}^{\pi,\theta}(s) \end{aligned}$$

- In all other cases we have

$$R_c^{\pi,\theta}(s) = A_c^{\pi,\theta}(s) = A_{c'}^{\pi,\theta}(s) = R_{c'}^{\pi,\theta}(s)$$

  by definition of the reads-set and Lemma 27. □

The instantiation requirement on $S^n_{\text{C-IL}}.reads$ then follows as a corollary.

PROOF OF $insta_r(S^n_{\text{C-IL}})$: Let $s^\pi(u,m) = (stmt_{next}(\pi,(\lceil m\rceil,u)))$. For reads-set

$$R = R^{\pi,\theta}_{(\lceil m\rceil,u)}(s^\pi(u,m))$$

and partial memories m, m' such that $m|_R = m'|_R$ (hence $\lceil m\rceil|_R = \lceil m'\rceil|_R$) we have by definition and Lemma 28:

$$\begin{aligned} S^n_{\text{C-IL}}.reads(u,m,in) &= R^{\pi,\theta}_{(\lceil m\rceil,u)}(s^\pi(u,m)) \overset{\text{L28}}{=} R^{\pi,\theta}_{(\lceil m'\rceil,u)}(s^\pi(u,m)) \\ &= R^{\pi,\theta}_{(\lceil m'\rceil,u)}(\pi.\mathcal{F}(u[|u|].f).P[u[|u|].loc]) \\ &= R^{\pi,\theta}_{(\lceil m'\rceil,u)}(s^\pi(u,m')) = S^n_{\text{C-IL}}.reads(u,m',in) \qquad \square \end{aligned}$$

5 Simulation in Concurrent Systems

Based on our order reduction theory we now want to explore how to apply local simulation theorems in a concurrent context. Our goal is to state and prove a global *Cosmos* model simulation theorem which argues that the local simulation theorems still hold on computation units. In particular we want the simulation relation to hold for a unit when it reaches a consistency point. Moreover from the verification of ownership safety on the higher level, memory safety on the lower level should follow. First we introduce a variation of the *Cosmos* model semantics tailored to the formulation of such a simulation theorem. Then we introduce sequential simulation theorems in a generalized manner. Building on the sequential theorems we then formulate and prove a concurrent simulation theorem between *Cosmos* machines, stating the necessary requirements on the sequential simulations to be composable with each other.

In the concurrent simulation we will profit from the *Cosmos* model order reduction theorem proven before. For every computation unit of the simulating *Cosmos* machine we set up the interleaving-points to be consistency points wrt. the sequential simulation relation. This enables us to conduct a simulation proof between $\mathcal{IP}$ schedules of *Cosmos* machines, applying the sequential simulation theorem separately on each $\mathcal{IP}$ block. In such a scenario, where the interleaving-points are also consistency points wrt. a given simulation relation we speak of *consistency blocks* instead of $\mathcal{IP}$ blocks.

Now the sequential simulation theorem can be applied on any consistency block on the simulated level in order to obtain the simulated abstract consistency block executed by the same unit. However there is a technicality to be solved, namely that the given concrete block may not be *complete* in the sense that it does not lead into another consistency point. Then one has to find an extension of that incomplete block, so that the resulting complete concrete block is simulating an abstract block. We have to formulate the generalized sequential simulation theorem in a way that allows for this kind of extension. Nevertheless later we will show for the transfer of verified safety properties that it suffices to consider schedules where each consistency block is complete.

5.1 Block Machine Semantics

Since we may assume $\mathcal{IP}$ schedules for safe *Cosmos* machine execution, semantics can be simplified. For introducing simulation theorems on *Cosmos* models it is convenient to define semantics where we consecutively execute blocks starting in interleaving-points ($\mathcal{IP}$ blocks). Also for now we do not need to consider ownership, therefore it is sufficient to model the transitions on the machine state. We call the machine implementing such semantics the *$\mathcal{IP}$ block machine* or short the *block machine*.

5.1.1 Definition

We define the block machine semantics for a *Cosmos* machine S. The block machine executes one $\mathcal{IP}$ block in a single step. To this end it gets a schedule $\kappa \in (\Theta_S^*)^*$ as a parameter which is a sequence of transition sequences representing the individual blocks to be executed. To distinguish blocks and block schedule we will always use λ for transition sequences and κ for block sequences. Naturally not all block sequences are valid block machine schedules. Each block in the block machine schedule needs to be an $\mathcal{IP}$ block.

Definition 77 ($\mathcal{IP}$ Block) *A transition sequence $\lambda \in \Theta_S^*$ is called an $\mathcal{IP}$ block of machine $p \in S.nu$ if it (i) contains only steps by that machine, (ii) is empty or starts in an interleaving-point, and (iii) does not contain any further interleaving-points.*

$$\begin{aligned} blk(\lambda, p) \equiv \quad & (i) \quad \forall \alpha \in \lambda.\ \alpha.s = p \\ & (ii) \quad \lambda \neq \varepsilon \implies \lambda_1.ip \\ & (iii) \quad \forall \alpha \in tl(\lambda).\ /\alpha.ip \end{aligned}$$

Thus we require the $\mathcal{IP}$ blocks to be *minimal* in the sense that they contain at most one interleaving-point. For technical reasons empty blocks are also considered $\mathcal{IP}$ blocks. We define the appropriate predicate $Bsched$ which denotes that a given a block sequence $\kappa \in (\Theta_S^*)^*$ is a block machine schedule.

$$Bsched(\kappa) \stackrel{def}{\equiv} \forall \lambda \in \kappa. \exists p \in \mathbb{N}_{nu}.\ blk(\lambda, p)$$

Note that this implies that the *flattening concatenation* $\lfloor \kappa \rfloor \stackrel{def}{\equiv} \kappa_1 \cdots \kappa_{|\kappa|}$ of all blocks of κ form an $\mathcal{IP}$ schedule.

Lemma 29 *The flattening concatenation of all blocks of any block machine schedule $\kappa \in (\Theta_S^*)^*$ is an $\mathcal{IP}$ schedule.*

$$Bsched(\kappa) \implies \mathcal{IP}sched(\lfloor \kappa \rfloor)$$

Instead of defining a transition function for the block machine we extend our step sequence notation to block sequences as follows.

Definition 78 (Step Notation for Block Sequences) *Given two machine states $M, M' \in \mathbb{M}_S$ and a block machine schedule $\kappa \in (\Theta_S^*)^*$, we denote that M' is reached by executing the block machine from state M wrt. schedule κ by the following notation.*

$$M \stackrel{\kappa}{\longmapsto} M' \equiv M \stackrel{\lfloor \kappa \rfloor}{\longmapsto} M'$$

Then a pair (M, κ) is a computation of the block machine, if there exists a machine state M' that can be reached via schedule κ from M, i.e., $M \stackrel{\kappa}{\longmapsto} M'$. Furthermore we need to introduce safety for the block machine wrt. the ownership policy and some safety property P. Similar to $safety$ and $safety_{\mathcal{IP}}$ defined earlier, the verification of all

block machine computations running out of configuration C wrt. ownership and some *Cosmos* machine safety property P is denoted by the following predicate.

$$safety_B(C,P) \stackrel{def}{\equiv} \forall \kappa \in (\Theta_S^*)^*.\ Bsched(\kappa) \wedge comp(C.M, \lfloor \kappa \rfloor) \implies \exists o \in \Omega_S^*.\ safe_P(C, \langle \lfloor \kappa \rfloor, o \rangle)$$

5.1.2 Reduction Theorem and Proof

In order to justify the verification of systems using block machine schedules instead of $\mathcal{IP}$ schedules, we need to give a reduction proof. However since the two concepts are so closely related this is a rather easy task. First we need to show that every $\mathcal{IP}$ schedule can be represented by a block machine schedule.

Lemma 30 (Block Machine Schedule Existence) *For any $\mathcal{IP}$ schedule θ we can find a corresponding block schedule κ such that the flattening concatenation of κ's blocks equals θ.*

$$\forall \theta \in \Theta_S^*.\ \mathcal{IP}sched(\theta) \implies \exists \kappa \in (\Theta^*)^*.\ Bsched(\kappa) \wedge \lfloor \kappa \rfloor = \theta$$

PROOF: by induction on $n = |\theta|$.

Induction Start: $n = 0$ - No step is made in θ and we set $\kappa = \varepsilon$. Then $\mathcal{IP}sched(\kappa)$ holds trivially since κ contains no blocks and also $\lfloor \kappa \rfloor = \varepsilon = \theta$.

Induction Hypothesis: For an n-step interleaving-point schedule $\bar{\theta}$ we can find a corresponding block machine schedule $\bar{\kappa}$ as claimed.

Induction Step: $n \to n+1$ - We split θ into $\bar{\theta}$ and α such that $\theta = \bar{\theta}\alpha$. By induction hypothesis we obtain $\bar{\kappa}$ which is a block machine schedule with $\lfloor \bar{\kappa} \rfloor = \bar{\theta}$. With $m = |\bar{\kappa}|$ we perform a case distinction on α.

1. $\alpha.ip \vee \bar{\theta}|_{\alpha.s} = \varepsilon$ - Step α starts a new $\mathcal{IP}$ block or the first block. We introduce $\lambda = \alpha$ is a transition sequence containing only α and fulfilling the properties $\forall \beta \in \lambda.\ \beta.s = \alpha.s$, $\lambda_1.ip$, and $tl(\lambda) = \varepsilon$. Therefore $blk(\lambda, \alpha.s)$ holds trivially and we obtain κ by simply adding λ to $\bar{\kappa}$ as a new block, i.e. $\kappa = \bar{\kappa}\lambda$. We immediately see from induction hypothesis and $blk(\lambda, \alpha.s)$ that:

   $$\forall \lambda' \in \kappa. \exists p \in \mathbb{N}_{nu}.\ blk(\lambda', p)$$

   Thus by definition we have $Bsched(\kappa)$. Moreover we have:

   $$\lfloor \kappa \rfloor = \lfloor \bar{\kappa}\lambda \rfloor = \bar{\kappa}_1 \ldots \bar{\kappa}_m \lambda = \lfloor \bar{\kappa} \rfloor \lambda = \bar{\theta}\lambda = \bar{\theta}\alpha = \theta$$

2. $\bar{\theta} \neq \varepsilon \wedge \alpha.s = \bar{\theta}_n.s \wedge / \alpha.ip$ - Step α simply extends the last $\mathcal{IP}$ block in $\mathcal{IP}$ schedule $\bar{\theta}$ which we denote by λ.

   $$\exists \lambda, \tau.\ \bar{\theta} = \tau\lambda \wedge blk(\lambda, \alpha.s)$$

The existence of such a λ is justified by induction hypothesis since in fact we have $\lambda = \bar{\kappa}_m$. Thus we can construct block machine schedule κ as follows.

$$\kappa[1:m) = \bar{\kappa}[1:m) \qquad \kappa_m = \lambda\alpha$$

Sequence λ is a well-formed $\mathcal{IP}$ block by induction hypothesis. In addition α is executed by the same unit as λ and it does not add an interleaving-point to the tail of λ. Therefore also $\lambda\alpha$ is wellformed and we have $\mathbb{B}sched(\kappa)$ by definition and IH on $\bar{\kappa}$. The second claim now holds with the following transformations.

$$\lfloor\kappa\rfloor = \lfloor\bar{\kappa}[1:m)\lambda\alpha\rfloor = \bar{\kappa}_1 \ldots \bar{\kappa}_{m-1}\lambda\alpha = \bar{\kappa}_1 \ldots \bar{\kappa}_m\alpha = \lfloor\bar{\kappa}\rfloor\alpha = \bar{\theta}\alpha = \theta$$

3. Since θ is an $\mathcal{IP}$ schedule, by definition the remaining case $\bar{\theta}_n.s \neq \alpha.s \wedge /\alpha.ip$ is not possible. □

Now we can easily reduce the verification of $\mathcal{IP}$ schedules to the verification of block machine computations. In particular we show that the verification of all block machine computations implies the verification of all $\mathcal{IP}$ schedules.

Theorem 4 (Block Machine Reduction) *Let C be a configuration of* Cosmos *machine S and P be a* Cosmos *machine safety property. Then if all block machine computations running out of C are ownership-safe and preserve P, the same holds for all $\mathcal{IP}$ schedules starting in C.*

$$safety_B(C,P) \implies safety_{\mathcal{IP}}(C,P)$$

PROOF: For the sake of contradiction we assume an $\mathcal{IP}$ schedule θ with $C.M \overset{\theta}{\longmapsto} M'$ that is unsafe, i.e., there is no ownership annotation for it that is obeying the ownerhsip policy or it breaks property P.

$$\mathcal{IP}sched(\theta) \qquad \forall o \in \Omega_S^*.\ /safe_P(C, \langle\theta, o\rangle)$$

By Lemma 30 shown above we know that there exists a corresponding equivalent block machine schedule κ such that $\lfloor\kappa\rfloor = \theta$. This immediately gives us $C.M \overset{\kappa}{\longmapsto} M'$ by definition of the block sequence notation. Moreover we can use $\lfloor\kappa\rfloor = \theta$ in the above statements obtaining

$$\exists\kappa \in (\Theta_S^*)^*.\ Bsched_B(\kappa) \wedge (\exists M'.\ C.M \overset{\kappa}{\longmapsto} M') \wedge \forall o \in \Omega_S^*.\ /safe_p(C, \langle\lfloor\kappa\rfloor, o\rangle)$$

which is equivalent to $/safety_B(C,P)$. This however contradicts our hypothesis hence there must exist a safe ownership annotation for θ. Therefore all interleaving-point schedules starting in C are safe. □

5.2 Generalized Sequential Simulation Theorems

As we have seen before, computer systems can be described on several layers of abstractions, e.g. on the ISA level, the levels of macro assembly and C-IL, or even higher levels of abstraction [GHLP05]. Between different levels there are sequential simulation theorems, as presented in the previous chapter. Such simulation theorems are proven for sequential execution traces where no environment steps are interleaved. However it is desirable to have the simulation relation hold also in the context of the concurrent system. Thus we need to be able to apply sequential simulation theorems in a system wide simulation proof between two *Cosmos* model instantiations $S_d, S_e \in \mathbb{S}$ where the interleaving-points are instantiated to be the consistency points wrt. the corresponding simulation relation. Recall that we speak of *consistency blocks* instead of $\mathcal{IP}$ blocks then.

In the sequel we develop a generalized theory of sequential simulation theorems. We consider the simulation between computations $(d, \sigma) \in \mathbb{M}_{S_d} \times \Theta^*_{S_d}$ and $(e, \tau) \in \mathbb{M}_{S_e} \times \Theta^*_{S_d}$ considering only the machine state of these *Cosmos* machines. We also speak of S_d as the *concrete* and of S_e as the *abstract* simulation layer, where computations of S_e are simulated by S_d. For simplicity we assume that the two systems have compatible memory types. Also both systems have the same number of computation units. Using the shorthands x_d and x_e for components $S_d.x$ and $S_e.x$ we demand:

$$\mathcal{A}_d \supseteq \mathcal{A}_e \qquad \mathcal{V}_d = \mathcal{V}_e \qquad nu_d = nu_e = nu$$

Observe that the memory address range of S_d might be larger than that of S_e. This means, that the latter may abstract from certain memory regions in the former. For example this is useful when we abstract a stack of local memories from a stack memory region when we consider compilation of C-IL programs as we have seen before. The stack region is then excluded from the shared memory. As we aim for a generalized theory about concurrent simulation theorems we first define a framework for specifying sequential simulation theorems in a uniform way.

Definition 79 (Sequential Simulation Framework) *We introduce a type $\mathbb{R}$ for simulation frameworks R_{S_d,S_e} which contain all the information needed to state a generalized simulation theorem relating sequential computations of units of* Cosmos *machines S_d and S_e.*

$$R_{S_d,S_e} = (\mathcal{P}, sim, \mathcal{CP}a, \mathcal{CP}c, wfa, sc, wfc, suit, wb) \in \mathbb{R}$$

In particular we have the following components where $\mathbb{L}_x \equiv (\mathcal{U}_x \times (\mathcal{A}_x \to \mathcal{V}_x))$ with $x \in \{d, e\}$ is a shorthand for the type of a local configuration *of* Cosmos *machine S_x containing the state of one computation unit and shared memory:*

- *$\mathcal{P}$ — the set of simulation parameters, which is $\{\bot\}$ if there are none,*
- *$sim : \mathbb{L}_d \times \mathcal{P} \times \mathbb{L}_e \to \mathbb{B}$ — a simulation relation between local configurations of computation units of S_d and S_e, depending on a simulation parameter from $\mathcal{P}$,*
- *$\mathcal{CP}a : \mathcal{U}_e \times \mathcal{P} \to \mathbb{B}$ — a predicate to identify consistency points of the abstract* Cosmos *machine S_e,*

- $\mathcal{CP}c : \mathcal{U}_d \times \mathcal{P} \to \mathbb{B}$ — *a predicate to identify consistency points of the concrete* Cosmos *machine* S_d*,*
- $wfa : \mathbb{L}_e \to \mathbb{B}$ — *a well-formedness condition for a local configuration of the abstract* Cosmos *machine* S_e*,*
- $sc : \mathbb{M}_{S_e} \times \Theta_{S_e} \times \mathcal{P} \to \mathbb{B}$ — *software conditions that enable a simulation of sequential computations of* Cosmos *machine* S_e*, here defined for a given step,*
- $wfc : \mathbb{L}_d \to \mathbb{B}$ — *well-formedness condition for a local configuration of the concrete* Cosmos *machine* S_d*, required for the simulation of sequential computations of* S_e*,*
- $suit : \Theta_{S_d} \to \mathbb{B}$ — *a predicate to determine whether a given step by the concrete* Cosmos *machine is suitable for simulation.*
- $wb : \mathbb{M}_{S_d} \times \Theta_{S_d} \times \mathcal{P} \to \mathbb{B}$ — *a predicate that restricts the simulating computations of* S_d*. We say that a simulating step in a computation of* S_d *is* well-behaved *iff it fulfills this restriction.*

Most of the components have their counterparts in the simulation theorems for MASM and C-IL defined earlier. Hence their purpose should be obvious. Since suitability and good behaviour (i.e., being well-behaved) were somewhat indiscriminate in these examples we again want to highlight the difference between the two concepts. While suitability is a necessary condition on the schedule of the concrete *Cosmos* machine for the simulation to work, good behaviour is a property that is guaranteed for simulating computations by the simulation theorem. These properties become important in a stack of simulation properties where they should imply the software conditions on the abstract layer of the underlying simulation theorem. See Section 6.1 for a more detailed discussion of the topic.

The consistency point predicates $\mathcal{CP}a$ and $\mathcal{CP}c$ are used later to define the interleaving-points in the concurrent *Cosmos* machine computations. Note that they depend only on the unit state and a simulation parameter, because in our examples for C-IL and MASM consistency points are determined independent of the state of memory. We could define $\mathcal{CP}a$ and $\mathcal{CP}c$ to also depend on memory, achieving a more general model, however doing so would complicate the proof of the concurrent simulation theorem,[1] hence we keep the model simple and omit the dependence on memory. Also, contrary to interleaving-points, consistency points are independent of external inputs.

Nevertheless we hold our framework to be suitable to cover sequential simulation theorems between a variety of systems. To get some intuition on the software conditions, well-formedness conditions, and restrictions on the behaviour of simulating computations that are involved in R_{S_d,S_e}, we will give an example by instantiating R_{S_d,S_e} with S_{MASM} and S_{MIPS}. Let $h = (c, m) \in \mathbb{L}_{\text{MIPS}}$, $l = (c, m) \in \mathbb{L}_{\text{MASM}}$, and

$$A_{cp}^{\text{MASM}} \equiv \{adr(info_\mu, p, loc) \mid p \in \text{dom}(\pi) \wedge loc \leq |\pi(p).body|\}$$

[1] One would need to argue that memory accesses of other units do not influence whether a given machine is in a consistency point.

in:

$$R_{S_{\mathrm{MIPS}},S_{\mathrm{MASM}}}.\begin{cases}\mathcal{P} & = & InfoT_{\mathrm{MASM}}\\ sim(h,info_\mu,l) & = & consis_{\mathrm{MASM}}(h,info_\mu,(l.c,\lceil l.m\rceil))\\ \mathcal{CP}a(c,info_\mu) & = & 1\\ \mathcal{CP}c(c,info_\mu) & = & (c.pc\in A^{\mathrm{MASM}}_{cp})\\ wfa(l) & = & wf_{\mathrm{MASM}}(l.c,\lceil l.m\rceil)\\ sc(M,t,info_\mu) & = & sc_{\mathrm{MASM}}((M.u(t.s),\lceil M.m\rceil),info_\mu)\\ wfc(h) & = & wf^{\mathrm{MASM}}_{\mathrm{MIPS}}(h)\\ suit(\alpha) & = & suit^{\mathrm{MASM}}_{\mathrm{MIPS}}(\alpha.in)\\ wb(M,t,info_\mu) & = & wb^{\mathrm{MASM}}_{\mathrm{MIPS}}((M.u(t.s),M.m),t.in)\end{cases}$$

In the sequential simulation of MASM computations by MIPS units, we need the assembler information as a simulation parameter and we naturally choose $consis_{\mathrm{MASM}}$ as the simulation relation. All configurations on the abstract MASM level are consistency points. On the concrete MIPS level we are in a consistency point iff the program counter points to the beginning of the assembled code for a MASM statement in the MASM program π. The remaining functions are simply instantiated using their counterparts from the MASM simulation theorem. We could give a similar instantiation of $R_{S_{\mathrm{MIPS}},S_{\text{C-IL}}}$ however we save this for the instantiation of the concurrent simulation theory to be introduced in the next section.

As mentioned before we need to be able to apply the sequential simulation theorem on incomplete consistency blocks. Thus we consider a given consistency block $\omega \in \Theta^*_{S_d}$ as the basis for the simulating concrete computation. We have to extend ω into a complete non-empty consistency block σ which is simulating some abstract consistency block. Formally the extension of some transition sequence is denoted by the relation $\omega \rhd^{blk}_p \sigma$ which is saying that σ extends ω without adding consistency points to the block. Alternatively we can say that ω is a prefix of consistency block σ

$$\omega \rhd^{blk}_p \sigma \stackrel{def}{\equiv} \exists\tau.\ \sigma = \omega\tau \neq \varepsilon \wedge blk(\sigma,p) \wedge blk(\omega,p)$$

In order to be able to integrate the sequential simulation theorems into the concurrent system later on, there is an additional proof obligation in the sequential simulation below. Basically it is there to justify the $\mathcal{IOIP}$ condition of the underlying order reduction theorem which demands that there is at most one $\mathcal{IO}$ step between two subsequent interleaving-points of the same computation unit. This property has to be preserved by the concrete implementation of the abstract specification level. Moreover there should be a one-to-one mapping of $\mathcal{IO}$ steps on the abstract level to $\mathcal{IO}$ steps on the concrete level. That means that in corresponding blocks *Cosmos* machine S_d may only perform an $\mathcal{IO}$ step when S_e does and vice versa. If this would not be the case we could not couple the ownership state of S_d and S_e later, because at $\mathcal{IO}$ steps we allow for ownership transfer. Transferring ownership on one level but not on the other then may lead to inconsistent ownership configurations. We denote the requirements on $\mathcal{IO}$ points in consistency blocks by the overloaded predicate $one\mathcal{IO}$. For a single transition sequence

σ it demands that σ contains only one $\mathcal{IO}$ step. For a pair (σ, τ) it demands that they contain the same number of $\mathcal{IO}$ steps but at most one.

$$one\mathcal{IO}(\sigma) \stackrel{def}{\equiv} \forall i, j \in \mathbb{N}_{|\sigma|}.\ \sigma_i.io \wedge \sigma_j.io \implies i = j$$

$$one\mathcal{IO}(\sigma, \tau) \stackrel{def}{\equiv} \begin{cases} \tau|_{io} = \varepsilon & : \quad \sigma|_{io} = \varepsilon \\ one\mathcal{IO}(\sigma) \wedge one\mathcal{IO}(\tau) & : \quad \text{otherwise} \end{cases}$$

Moreover we introduce the following shorthands for $d \in \mathbb{M}_{S_d}$, $e \in \mathbb{M}_{S_e}$, $p \in \mathbb{N}_{nu}$, $par \in R_{S_d,S_e}.\mathcal{P}$, $\omega \in \Theta_{S_d}$, and $\tau \in \Theta_{S_e}$.

$$\begin{aligned}
\mathcal{P} &\equiv R_{S_d,S_e}.\mathcal{P} \\
sim_p(d, par, e) &\equiv R_{S_d,S_e}.sim((d.u(p), d.m), par, (e.u(p), e.m)) \\
\mathcal{CP}_p(e, par) &\equiv R_{S_d,S_e}.\mathcal{CP}a(e.u(p), par) \\
\mathcal{CP}_p(d, par) &\equiv R_{S_d,S_e}.\mathcal{CP}c(d.u(p), par) \\
wf_p(e) &\equiv R_{S_d,S_e}.wfa(e.u(p), e.m) \\
sc(e, \tau, par) &\equiv \forall \theta, \alpha, \theta', e'.\ \tau = \theta\alpha\theta' \wedge e \stackrel{\theta}{\longmapsto} e' \implies R_{S_d,S_e}.sc(e', \alpha, par) \\
wf_p(d) &\equiv R_{S_d,S_e}.wfc(d.u(p), d.m) \\
suit(\omega) &\equiv \forall \alpha \in \omega.\ R_{S_d,S_e}.suit(\alpha) \\
wb(d, \omega, par) &\equiv \forall \theta, \alpha, \theta, d'.\ \omega = \theta\alpha\theta' \wedge d \stackrel{\theta}{\longmapsto} d' \implies R_{S_d,S_e}.wb(d', \alpha, par)
\end{aligned}$$

Note that we overload $\mathcal{CP}_p$ and use both for machine states of type $\mathbb{M}_{S_d}$ and $\mathbb{M}_{S_e}$. In the same way we have overloaded wf_p. In what follows we will always use letter d to represent concrete machine states and letter e for abstract ones.

Now the generalized sequential simulation theorem is formulated along the lines of the MIPS-MASM and the MIPS-C-IL simulation theorems that we have seen in the previous chapter. However there are two technical differences. First, we use the our step notation instead of talking about sequences of machine configurations so that it fits with our order reduction theory and we can apply it on the consistency blocks of concrete block machine computations.

Secondly, the theorem is stated such that it allows for completing incomplete consistency blocks on the concrete abstraction layer. Given a concrete machine computation (d, ω), where the simulation relation sim_p holds between initial machine state d and an abstract state e for some computation unit p and ω is an incomplete consistency block executed by p. We need to be able to extend ω into a transition sequence σ that leads into a consistency point, obtaining a complete consistency block for which there is a simulated computation (e, τ) on the abstract level (cf. Fig. 20).

The ability to extend incomplete consistency into complete ones is important in the proof of the concurrent simulation theorem where we need to find a simulated abstract computation for a concurrent concrete block machine computation, where most of the consistency blocks are probably incomplete. In this situation we can use the generalized sequential simulation theorem for completing the concrete blocks and finding the simulated abstract consistency blocks. Formally the theorem reads as follows.

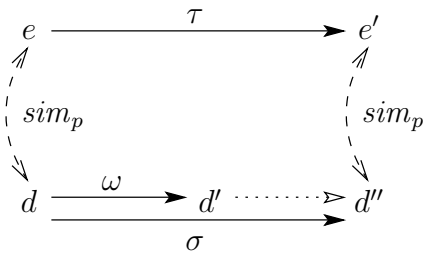

Figure 20: Illustration of the generalized sequential simulation theorem. Here σ extends consistency block ω of unit p, i.e., $\omega \rhd_p^{blk} \sigma$, such that the computation reaches another consistency point and simulates abstract computation (e, τ).

Theorem 5 (Generalized Sequential Simulation Theorem) *Given are two starting machine states $d \in \mathbb{C}_{S_d}$, $e \in \mathbb{C}_{S_e}$, a simulation parameter $par \in R_{S_d,S_e}.\mathcal{P}$ and a transition sequence $\omega \in \Theta^*_{S_d}$. If for any computation unit $p \in \mathbb{N}_{nu}$ (i) d and e are well-formed and (ii) consistent wrt. par, (iii) ω is a possibly incomplete consistency block of unit p that is suitable for simulation and executable from d, and (iv) all complete consistency blocks of unit p which are starting in e are obeying the software conditions for S_e and lead into well-formed configurations,*

$$\begin{array}{lll} \forall d, e, par, \omega, p. & \textit{(i)} & wf_p(d) \wedge wf_p(e) \\ & \textit{(ii)} & sim_p(d, par, e) \wedge \mathcal{CP}_p(d, par) \wedge \mathcal{CP}_p(e, par) \\ & \textit{(iii)} & blk(\omega, p) \wedge suit(\omega) \wedge \exists d'.\ d \overset{\omega}{\longmapsto} d' \\ & \textit{(iv)} & \forall \pi, e'.\ e \overset{\pi}{\longmapsto} e' \wedge blk(\pi, p) \wedge \mathcal{CP}_p(e', par) \implies sc(e, \pi, par) \wedge wf_p(e') \end{array}$$

then we can find sequences $\sigma \in \Theta^*_{S_d}$, $\tau \in \Theta^*_{S_e}$ and configurations $d' \in \mathbb{C}_{S_d}$, $e'' \in \mathbb{C}_{S_e}$ such that (i) σ is a suitable schedule and a consistency block of unit p extending the given block ω, τ is a consistency block of unit p, and σ and τ contain the same amount of $\mathcal{IO}$ steps but at most one. Moreover (ii) (d, σ) is a well-behaved computation with leading into well-formed state d'' and (iii) executing τ from e leads into well-formed configuration e''. Finally (iv) d'' and e'' are consistency points of unit p and consistent wrt. simulation parameter par:

$$\begin{array}{lll} \implies \quad \exists \sigma, \tau, d'', e''. & \textit{(i)} & \omega \rhd_p^{blk} \sigma \wedge suit(\sigma) \wedge blk(\tau, p) \wedge one\mathcal{IO}(\sigma, \tau) \\ & \textit{(ii)} & d \overset{\sigma}{\longmapsto} d'' \wedge wb(d, \sigma, par) \wedge wf_p(d'') \\ & \textit{(iii)} & e \overset{\tau}{\longmapsto} e'' \wedge wf_p(e'') \\ & \textit{(iv)} & sim_p(d'', par, e'') \wedge \mathcal{CP}_p(d'', par) \wedge \mathcal{CP}_p(e'', par) \end{array}$$

Note that for the simulated computation (e, τ) we only demand progress (i.e., $\tau \neq \varepsilon$) in case σ contains $\mathcal{IO}$ steps. Then $\tau \neq \varepsilon$ follows from $one\mathcal{IO}(\sigma, \tau)$. In contrast, by $\omega \rhd_p^{blk} \sigma$ we only consider such computations (d, σ) that are progressing in every simulation step, i.e., $\sigma \neq \varepsilon$. This setting rules out trivial simulations with empty transition sequences σ and τ in case $\omega = \varepsilon$.

As noted before, the formulation above does not match one to one the style of the sequential simulation theorems for MASM and C-IL in the previous chapter. This is mainly due to the fact that here we are using the convenient step sequence notation and that in the previous theorems we did not consider incomplete consistency blocks as the base of our simulation theorem. Nevertheless, if we set $\omega = \varepsilon$, instantiate the predicates of simulation framework adequately, and adjust the notation to the state sequence style, we can see that these theorems are in fact special cases of the one defined above.

One might wonder how a simulation theorem like the one above is actually proven. If we consider for instance the MASM instantiation from above, an important proof obligation is the requirement that only code from the code region is fetched, i.e., that only the compiled code of the MASM program is executed. Consequently (d, ω) is part of the MIPS implementation of some MASM computation (e, τ), i.e., it is computing the compiled program code. Since the MASM code generation function produces a finite amount of code for each MASM instruction, we know that if the processor keeps on executing the code without interruption, control will finally arrive in a consistency point again. Then one just needs to argue about the correctness of the generated code in order to prove the simulation of (e, τ).

Another notable property of our generalized sequential simulation theorem is that we support only static simulation parameters. There are certain simulation theorems which depend on a sequence of simulation parameters that is created "dynamically" during the simulation. For instance, in a programming language with explicite heap, in the simulation we would need to keep track of the heap allocation function as a simulation parameter that is changing dynamically during the execution of the program. While including such a parameter sequence in the theorem presented above can easily be done, presenting a use case for dynamic simulation parameters would be beyond the scope of this thesis. Therefore we leave this extension as future work

Finally, note that the sequential simulation theorem does not restrict the ownership state in any way. All predicates depend only on the machine state of a *Cosmos* machine. However for proving our concurrent simulation theorem, we will need an assumption on the ownership-safety of the simulated computation.

5.3 *Cosmos* Model Simulation

Using the sequential simulation theorems in an interleaved execution trace, we now aim to establish a system-wide simulation between two block machine computations (d, κ) and (e, ν). The simulating (concrete) computation (d, κ) need not be complete. However (e, ν) is a complete block machine computation. In Section 5.5 we will reduce reasoning to simulation between *complete block machine computations*.

5.3.1 Consistency Blocks and Complete Block Machine Computations

We already introduced the notions of complete and incomplete consistency blocks informally. Now we want to give a formal definition. Consistency blocks start in consis-

tency points, i.e. configurations of S_d in which the sequential simulation relation holds wrt. some configuration of S_e and vice versa. Our concurrent simulation theorem is based on the application of our order reduction theorem on S_d where we choose the interleaving-points to be exactly the consistency points as mentioned before. Similarly interleaving-points and consistency points of S_e are identical. These requirements on the instantiation of S_d and S_e are formalized in the following predicate.

Definition 80 (Interleaving-Points are Consistency Points) *Given a sequential simulation framework R_{S_d,S_e} which relates two* Cosmos *machines S_d and S_e and a simulation parameter $par \in \mathcal{P}$ we define a predicate denoting that in S_d and S_e the interleaving-points are set up to be exactly the consistency points.*

$$\begin{array}{rcl} \mathcal{IPCP}(R_{S_d,S_e}, par) & \equiv & \forall d \in \mathbb{M}_{S_d}, \alpha \in \Theta_{S_d}.\ \ \mathcal{IP}_{\alpha.s}(d, \alpha.in) \Longleftrightarrow \mathcal{CP}_{\alpha.s}(d, par) \\ & \wedge & \forall e \in \mathbb{M}_{S_e}, \beta \in \Theta_{S_e}.\ \ \mathcal{IP}_{\beta.s}(e, \beta.in) \Longleftrightarrow \mathcal{CP}_{\beta.s}(e, par) \end{array}$$

If these properties holds we speak of consistency blocks instead of $\mathcal{IP}$ blocks. This is reflected in the definition of consistency block machine schedules $\kappa \in (\Theta^*_{S_d})^* \cup (\Theta^*_{S_e})^*$.

$$\mathcal{CP}sched(\kappa, par) \ \equiv \ Bsched(\kappa) \wedge \mathcal{IPCP}(R_{S_d,S_e}, par)$$

Given a *Cosmos* machine state $d \in \mathbb{C}_{S_d}$ and a simulation parameter par as above we can define the set U_c of computation units of d that are currently in consistency points wrt. the simulation parameter par.

$$U_c(d, par) \ \equiv \ \{p \in \mathbb{N}_{S_d.nu} \mid \mathcal{CP}_p(d, par)\}$$

With the above setting of interleaving-points for par thus for any computation (d, α) with $\alpha.ip$ we have $\alpha.s \in U_c(d, par)$. Now a complete block machine computation is a block machine computation where all computation units are in consistency points in every configuration. This is encoded in the following overloaded predicate.

$$\begin{array}{rcl} \mathcal{CP}sched_c(d, \kappa, par) & \equiv & \mathcal{CP}sched(\kappa, par) \wedge \forall \kappa', \kappa'', d'. \\ & & \qquad \kappa = \kappa'\kappa'' \wedge d \stackrel{\kappa'}{\longmapsto} d' \implies \forall p \in \mathbb{N}_{nu}.\, \mathcal{CP}_p(d', par) \\ \mathcal{CP}sched_c(e, \nu, par) & \equiv & \mathcal{CP}sched(\nu, par) \wedge \forall \nu', \nu'', e'. \\ & & \qquad \nu = \nu'\nu'' \wedge e \stackrel{\nu'}{\longmapsto} e' \implies \forall p \in \mathbb{N}_{nu}.\, \mathcal{CP}_p(e', par) \end{array}$$

Note that we could prove the reduction of arbitrary consistency block machine schedules to complete ones given that for every machine it is always possible to reach a consistency point again (completability). However the completability assumption needs to be justified by the simulation running on the machine. In addition, the consistency points are only meaningful in connection with a simulation theorem. Thus it is useless to treat the reduction of incomplete blocks on a single layer of abstraction. The safety transfer theorem for complete block schedules along with our *Cosmos* model simulation theory will be presented in the subsequent sections. There the verification

of ownership-safety and a *Cosmos* model safety property P for all complete block machine schedules running out of a configuration $C \in \mathbb{C}_{S_d} \cup \mathbb{C}_{S_e}$ is defined below with $\Omega = \Omega_{S_d} = \Omega_{S_e}$ and $\Theta = \Theta_{S_d} \cup \Theta_{S_e}$.

$$safety_{cB}(C, P, par) \stackrel{def}{\equiv} \forall \kappa \in (\Theta^*)^*.\ \mathcal{CP}sched_c(C, \kappa, par) \wedge comp(C.M, \lfloor \kappa \rfloor) \implies \exists o \in \Omega^*.\ safe_P(C, \langle \lfloor \kappa \rfloor, o \rangle)$$

5.3.2 Requirements on Sequential Simulation Relations

Now we define the overall simulation relation between two machine states $d \in \mathbb{C}_{S_d}$ and $e \in \mathbb{C}_{S_e}$. Basically we demand that the local simulation relations hold for all machines in consistency points.

$$sim(d, par, e) \stackrel{def}{\equiv} \forall p \in U_c(d, par).\ sim_p(d, par, e)$$

We will later on require that the simulation relation holds between the corresponding machine states of the consistent *Cosmos* machine computations. This means that there are units in the concrete computation which are at times not coupled with the computation on the abstract simulation layer. More precisely, this is the case for units which have not reached a consistency point again at the end of the computation, i.e., their last block in the block machine schedule is incomplete. Only for complete block machine computations we have that units are coupled in all intermediate machine states. In order to compose the simulations we assume a certain structure and properties of the simulation relations which enable the composition in the first place. We introduce the following framework for concurrent simulation between S_d and S_e.

Definition 81 (Concurrent Simulation Framework) *A concurrent simulation framework for* Cosmos *machines S_d and S_e is a pair containing the sequential simulation framework R_{S_d,S_e} as well as a* shared memory and ownership invariant *sinv (short: shared invariant) that is coupling and constraining the shared memory and the ownership states of both systems. Let $\mathcal{M}_x = \mathcal{A}_x \rightharpoonup \mathcal{V}_x$ and $\mathbb{O}_x = \mathbb{N}_{nu} \to 2^{\mathcal{A}_x}$ in:*

$$sinv : (\mathcal{M}_d \times 2^{\mathcal{A}_d} \times 2^{\mathcal{A}_d} \times \mathbb{O}_d) \times \mathcal{P} \times (\mathcal{M}_e \times 2^{\mathcal{A}_e} \times 2^{\mathcal{A}_e} \times \mathbb{O}_e) \to \mathbb{B}$$

We introduce a shorthand that is asserting the shared invariant on two Cosmos *machine configurations D and E. Let $G_x(C) = (C.m|_{C.\mathcal{S} \cup S_x.\mathcal{R}}, C.\mathcal{S}, S_x.\mathcal{R}, C.\mathcal{G}.\mathcal{O})$ in:*

$$sinv(D, par, E) \equiv sinv(G_d(D), par, G_e(E))$$

Recall here that $C.\mathcal{G}.\mathcal{O}$ is the mapping of units to ownership sets that is part of the *Cosmos* machine ghost state. Note also that the $sinv(D, par, E)$ depends only on the ownership state and the portion of memory covered by the shared addresses. Thus ownership-safe local steps are preserving the shared invariant, since by the ownership-policy they do not modify the ownership state nor shared memory.

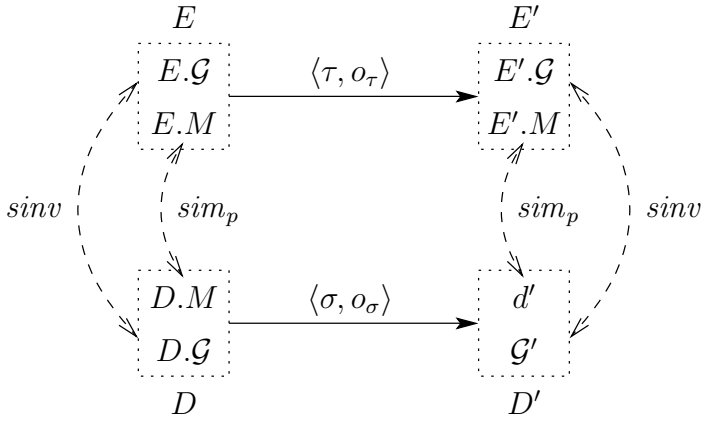

Figure 21: Illustration of Assumption 1. The simulating computation $\langle \sigma, o_\sigma \rangle$ must be ownership-safe and preserve the shared invariant $sinv$.

The shared invariant is introduced as a common abstraction relation on the shared memory and ownership model of S_d and S_e. If $\mathcal{A}_d = \mathcal{A}_e$, then $sinv$ should be just an identity mapping between the corresponding components of the concrete and abstract simulation levels. However, as we allow to abstract from portions of the abstract memory, the shared invariant may be more complex.

For instance in the C-IL scenario we abstract the function frames from the stack region in memory. While these memory regions are invisible on the abstract level, we would like to protect them via the ownership model from modification by other threads on the concrete level. Moreover there may be an abstraction of other shared resources between the two simulation layers we are considering.

The shared invariant is then used to cover such resource abstraction relations and formulate instantiation-specific ownership invariants on the concrete level of abstraction. We will give examples for the shared invariant later when we instantiate the concurrent simulation framework with MASM and C-IL. Below we formulate constraints on the predicates and the simulation relation introduced above, needed for a successful integration of the sequential simulation theorems into a concurrent one. These assumptions must be discharged by any instantiation of the concurrent simulation framework.

The most important assumption is stated first. On the one hand we require computation units of S_d and S_e to maintain $sinv$ according to the software conditions on computations of S_e and the definition of good behaviour for computations of S_d.

Moreover we need to assume an ownership-safety transfer theorem about the simulation which is essential in the construction of a pervasive concurrent model stack using ownership-based order reduction. While we abstract from computations bottom-up, the ownership-safety has to be verified on the top level and transfered downwards. For each abstraction level we have to show that ownership-safety is preserved by the implementation. These requirements are formalized as follows.

Assumption 1 (Safety Transfer and $sinv$ Preservation) *Consider a concurrent simulation framework $(R_{S_d,S_e}, sinv)$ and a complete consistency block computation $(D.M, \sigma)$ that is implementing an abstract consistency block $(E.M, \tau)$. We assume that (i) the concrete computation is well-behaved, leading into state $d' \in \mathbb{M}_{S_d}$, σ is a consistency block of p, and both schedules contain the same number of $\mathcal{IO}$ steps but at most one. Moreover (ii) τ is also a consistency block of p, the computation is safe according to ownership annotation o_τ, and leads into $E' \in \mathbb{M}_{S_e}$ obeying the software conditions on S_e. Finally (iii) the simulation relation for p and the shared invariant holds between $D.M$ and $E.M$, and the simulation relation holds also for the resulting configurations.*

$$
\begin{array}{ll}
\forall D, d', E, E', \sigma, \tau, o_\tau, p, par. & \\
\quad \textit{(i)} & D.M \overset{\sigma}{\longmapsto} d' \wedge blk(\sigma, p) \wedge one\mathcal{IO}(\sigma, \tau) \wedge wb(D.M, \sigma, par) \\
\quad \textit{(ii)} & E \overset{\langle \tau, o_\tau \rangle}{\longmapsto} E' \wedge blk(\tau, p) \wedge safe(E, \langle \tau, o_\tau \rangle) \wedge sc(E.M, \tau, par) \\
\quad \textit{(iii)} & sim_p(D.M, par, E.M) \wedge sinv(D, par, E) \wedge sim_p(d', par, E'.M)
\end{array}
$$

Then there exists an ownership annotation o_σ for σ, such that the annotated concrete computation (i) results in d' and a ghost state $\mathcal{G}'$, (ii) it is ownership-safe, and (iii) preserves $sinv$.

$$
\begin{array}{lll}
\Longrightarrow & \exists o_\sigma \in \Omega^*_{S_d}, \mathcal{G}'. & \textit{(i)} \quad D \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d', \mathcal{G}') \\
& & \textit{(ii)} \quad safe(D, \langle \sigma, o_\sigma \rangle) \\
& & \textit{(iii)} \quad sinv((d', \mathcal{G}'), par, E')
\end{array}
$$

See Fig. 21 for an illustration. For MASM the ownership-safety transfer requirement implies, e.g., that the code generated for macros may not access shared memory, because macros are not $\mathcal{IO}$ steps by definition. In case of C-IL in order to discharge the assumption we would need to show, e.g., that volatile accesses are compiled correctly such that the correct addresses are accessed. Additionally we would need to prove that the memory accesses implementing stack operations are only targeting the stack region and that ownership on the concrete level can be set up such that these memory accesses are safe. Again we refer to Sect. 5.6 for more examples.

Note that above we do not restrict in any way the ownership transfer on S_e. This means conversely that $sinv$ can in fact only restrict the ownership state of S_d that is not covered by $\mathcal{A}_e$. Moreover, assumption $safe(E, \langle \tau, o_\tau \rangle)$ and the shared invariant between D and E imply $inv(D)$. The sequential simulation relation does not cover the ownership state but is needed for technical reasons, too. We show this as a corollary.

Corollary 2 *If two ghost configurations $\mathcal{G}_d$ and $\mathcal{G}_e$ are coupled by the shared invariant and the simulation relation for any p, then the ownership invariant is transfered from $\mathcal{G}_e$ to $\mathcal{G}_d$.*

$$(\exists M_d, M_e.\ sinv((M_d, \mathcal{G}_d), par, (M_d, \mathcal{G}_e)) \wedge sim_p(M_d, par, M_e)) \wedge inv(\mathcal{G}_e) \implies inv(\mathcal{G}_d)$$

PROOF: By $\sigma = \tau = \varepsilon$ the hypotheses of Assumption 1 applied for $D = (M_d, \mathcal{G}_d)$ and $E = (M_e, \mathcal{G}_e)$ collapse to $sim_p(D.M, par, E.M)$, $sinv(D, par, E)$ and $inv(E)$ which hold by our hypothesis. Thus we have $safe(D, \varepsilon)$ which in turn implies $inv(D)$. □

Below we introduce another property which is needed to establish the sequential consistency relations in a concurrent setting.

Assumption 2 (Preservation of sim_p) *The sequential simulation relation for unit p only depends on p's local state and the memory covered by the shared invariant.*

$$\begin{aligned}\forall D, D' \in \mathbb{C}_{S_d}, E, E' \in \mathbb{C}_{S_e}, par \in \mathcal{P}, p \in \mathbb{N}_{nu}.\\ sim_p(D.M, par, E.M) \wedge D \approx_p D' \wedge E \approx_p E' \wedge sinv(D', par, E')\\ \implies sim_p(D'.M, par, E'.M)\end{aligned}$$

This assumption allows us to maintain simulation during environment steps. Furthermore the well-formedness of machine states cannot be broken by safe steps of other participants in the system if they maintaining the shared invariant.

Assumption 3 (Preservation of Well-formedness) *The well-formedness predicates only depend on the local state of their respective units and the memory covered by the shared invariant. For all $D, D' \in \mathbb{C}_{S_d}$, $E, E' \in \mathbb{C}_{S_e}$, $par \in \mathcal{P}$, and $p \in \mathbb{N}_{nu}$ we have:*

$$\begin{aligned}wf_p(D.M) \wedge D \approx_p D' \wedge sinv(D', par, E') &\implies wf_p(D'.M)\\ wf_p(E.M) \wedge E \approx_p E' \wedge sinv(D', par, E') &\implies wf_p(E'.M)\end{aligned}$$

5.3.3 Simulation Theorem

With the assumptions stated above we can show a global *Cosmos* model simulation theorem, given computations on the abstract level are proven to be safe wrt. ownership and a *Cosmos* machine safety property P. We claim that it is enough to verify all complete block computations leaving starting state E. This is the crucial prerequesite to enable a safe composition of computations. From a given consistency point a sequential computation of some unit p into the next consistency point must be safe. We do not treat property transfer for other safety properties than ownership-safety for now. However we instantiate the *Cosmos* machine safety property P so that it implies the well-formedness of machine states of S_e and that computations obey the software conditions of the abstract simulation layer.

Since obeying the software conditions is a property of steps rather than of states, we extend the unit states of S_e with some history information, recording the occurrence of software condition violations. Thus we use a modified *Cosmos* machine S'_e where each unit gets an additional boolean flag sc which is initially 1 and becomes 0 as soon as a step violates the software conditions, i.e., for all $\alpha \in \Theta_{S'_e}$, $e, e' \in \mathbb{M}_{S'_e}$, $par \in R_{S_d,S_e}.\mathcal{P}$ and $p \in \mathbb{N}_{S_e.nu}$ we have:

$$e \overset{\alpha}{\mapsto} e' \implies e'.u(p).sc = e.u(p).sc \wedge sc(e, \alpha, par)$$

Assuming the generalized sequential simulation theorem to be proven and the simulation relations and predicates to be constrained as presented above, we can now show the desired concurrent simulation theorem.

Theorem 6 (*Cosmos* Model Simulation Theorem) *Given are two* Cosmos *machine start configurations $D \in \mathbb{C}_{S_d}$ and $E \in \mathbb{C}_{S_e}$ as well as block machine schedule κ and concurrent simulation framework $(R_{S_d,S'_e}, sinv)$. We assume that (i) κ is a suitable consistency block schedule without empty blocks, (ii) that κ is executable from $D.M$ and at least one machine in D is in a consistency point, that (iii) all complete block machine computations running out of E are proven to obey ownership-safety and maintain* Cosmos *machine property P, and that (iv) P implies that every computation unit of S'_e is well-formed and does not violate software conditions. Moreover (v) units of D in consistency-points are well-formed. Finally we require that (vi) D and E are consistent wrt. simulation parameter $par \in \mathcal{P}$ and the shared invariant holds.*

$$
\begin{array}{lll}
\forall D, \kappa, E, par, P. & \textit{(i)} & \mathcal{CP}sched(\kappa, par) \wedge \forall \lambda \in \kappa.\ \lambda \neq \varepsilon \wedge suit(\lambda) \\
& \textit{(ii)} & comp(D.M, \lfloor \kappa \rfloor) \wedge \exists p \in \mathbb{N}_{nu}.\ \mathcal{CP}_p(D.M, par) \\
& \textit{(iii)} & safety_{cB}(E, P, par) \\
& \textit{(iv)} & \forall E' \in \mathbb{C}_{S'_e}, p \in \mathbb{N}_{nu}.\ P(E') \implies wf_p(E'.M) \wedge E'.u_p.sc \\
& \textit{(v)} & \forall p \in U_c(D.M, par).\ wf_p(D.M) \\
& \textit{(vi)} & sim(D.M, par, E.M) \wedge sinv(D, par, E)
\end{array}
$$

If these hypotheses hold we can show that there exists a block machine schedule ν such that (i) ν is complete, has the same length as κ, and describes a Cosmos *machine computation starting in $E.M$. This computation is simulated by $(D.M, \kappa)$ and for the resulting machine states M'_d and M'_e we know that (ii) they are well-formed for all units of M'_e and for all units M'_d in consistency points, and (iii) the simulation relation. Moreover (iv) the s imulating computationand well-behaved and each corresponding pair of consistency blocks contains the same number of $\mathcal{IO}$ steps but at most one. Finally (v) for any ownership annotation $o_\nu \in \Omega^*_{S_e}$ to computation $(E.M, \nu)$ that is safe and producing a ghost state $\mathcal{G}'_e$, we can find a corresponding annotation $o_\kappa \in \Omega^*_{S_d}$ for $(D.M, \kappa)$ resulting (v.a) in ghost state $\mathcal{G}'_d$ such that (v.b) the computation is ownership-safe and (v.c) the shared invariant holds between the resulting* Cosmos *machine configurations.*

$$
\begin{array}{lll}
\exists \nu, M'_d, M'_e. & \textit{(i)} & \mathcal{CP}sched_c(E.M, \nu, par) \wedge |\nu| = |\kappa| \wedge D.M \overset{\kappa}{\longmapsto} M'_d \wedge E.M \overset{\nu}{\longmapsto} M'_e \\
& \textit{(ii)} & \forall p \in \mathbb{N}_{nu}.\ wf_p(M'_e) \wedge \forall p \in U_c(M'_d, par).\ wf_p(M'_d) \\
& \textit{(iii)} & sim(M'_d, par, M'_e) \\
& \textit{(iv)} & wb(D.M, \lfloor \kappa \rfloor, par) \wedge \forall j \leq |\kappa|.\ one\mathcal{IO}(\kappa_j, \nu_j) \\
& \textit{(v)} & \forall o_\nu, \mathcal{G}'_e.\ E \overset{\langle \lfloor \nu \rfloor, o_\nu \rangle}{\longmapsto} (M'_e, \mathcal{G}'_e) \wedge safe(E, \langle \lfloor \nu \rfloor, o_\nu \rangle) \implies \\
& & \quad \exists o_\kappa, \mathcal{G}'_d. \quad \textit{(v.a)} \quad D \overset{\langle \lfloor \kappa \rfloor, o_\kappa \rangle}{\longmapsto} (M'_d, \mathcal{G}'_d) \\
& & \qquad\qquad\ \ \textit{(v.b)} \quad safe(D, \langle \lfloor \kappa \rfloor, o_\kappa \rangle) \\
& & \qquad\qquad\ \ \textit{(v.c)} \quad sinv((M'_d, \mathcal{G}'_d), par, (M'_e, \mathcal{G}'_e))
\end{array}
$$

The simulation theorem is illustrated in Fig. 22. Note that we do not require that all units start in consistency blocks, however this is implicitly guaranteed for all units running in κ by the definition of block machine schedules and the $\mathcal{IPCP}$ condition. If $\kappa = \varepsilon$ then hypothesis (ii) ensures that sim does not hold vacuously between D and E.

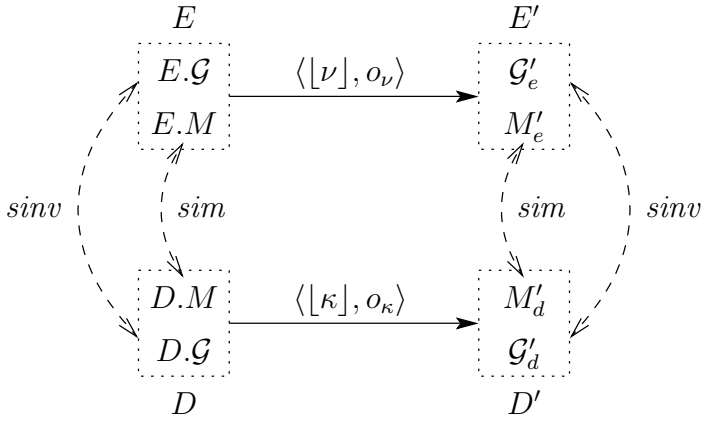

Figure 22: The *Cosmos* model simulation theorem. Computation $(D, \langle \lfloor \kappa \rfloor, o_\kappa \rangle)$ is ownership-safe and simulates an abstract computation $(E, \langle \lfloor \nu \rfloor, o_\nu \rangle)$.

Furthermore, in the simulation theorem a possibly incomplete consistency block machine computation of S_d is simulating a complete consistency block machine computation by S'_e. For the computation units whose final blocks are incomplete, i.e., who have not yet reached another consistency point, the simulation relation is not holding. However in all intermediate states of the block machine computation the shared invariant must hold. For the treatment of incomplete blocks we thus distinguish two cases.

On one hand, if the incomplete block contains only local steps we can simply omit it and represent it by a stuttering step (i.e., an empty block) on the abstract simulation level, because it does not affect the shared memory or ownership state.

On the other hand, if the incomplete block contains an $\mathcal{IO}$ step it may affect the shared memory or ownership state and in order to maintain the shared invariant the incomplete block must be represented properly on the abstract level. To this end we use the sequential simulation relation completing the block and obtaining the simulated consistency block of the abstract *Cosmos* machine computation. These are the core ideas of the proof of the concurrent simulation.

Note that we need to find a safe annotation for $(D.M, \kappa)$ for any given safe annotation on the abstract level. It does not suffice to simply find one pair of safe annotations for the simulating computations, because such a formulation is not applicable in the inductive proof of ownership-safety transfer.

5.3.4 Auxiliary Lemmas

Before we present the proof of Theorem 6 we need to show two useful lemmas.

Lemma 31 (Safe Local Steps Preserve Shared Invariant) *Given* Cosmos *machine configurations $E \in \mathbb{C}_{S'_e}$ and $D, D' \in \mathbb{C}_{S_d}$ where D' is reached from D by step sequence σ which contains no $\mathcal{IO}$ steps, as well as a simulation parameter par. Assuming that (D, σ) is safe,*

then D and E are coupled by the shared invariants iff D' and E are.

$$D \overset{\sigma}{\longmapsto} D' \wedge \sigma|_{io} = \varepsilon \wedge \mathit{safe}(D, \sigma) \implies (\mathit{sinv}(D, \mathit{par}, E) \Leftrightarrow \mathit{sinv}(D', \mathit{par}, E))$$

PROOF: by induction on $n = |\sigma|$. For $n = 0$, $D = D'$ holds and the equivalence is obvious. In the induction step from $n \to n+1$ we divide σ into prefix ω and last step α:

$$\sigma = \omega\alpha \qquad\qquad D \overset{\omega}{\longmapsto} D'' \overset{\alpha}{\mapsto} D'$$

By induction hypothesis on ω we have:

$$\mathit{sinv}(D, \mathit{par}, E) \Leftrightarrow \mathit{sinv}(D'', \mathit{par}, E)$$

From hypothesis we know that (D'', α) is safe, hence we can apply Lemma 19 with $C := D''$ and $C' := D'$ obtaining $D'' \approx_{\overline{q}} D'$ where $q = \alpha.s$. The definition of $\approx_{\overline{q}}$ then implies:

$$D''.m|_{D''.\mathcal{S} \cup S_d.\mathcal{R}} = D'.m|_{D''.\mathcal{S} \cup S_d.\mathcal{R}} \qquad D''.\mathcal{S} = D'.\mathcal{S} \qquad \forall p \in \mathbb{N}_{S_d.nu}.\ D''.\mathcal{O}_p = D'.\mathcal{O}_p$$

Therefore by definition also $G_d(D'') = G_d(D')$ and we conclude:

$$\begin{aligned} \mathit{sinv}(D, \mathit{par}, E) &\Leftrightarrow \mathit{sinv}(D'', \mathit{par}, E) \\ &\Leftrightarrow \mathit{sinv}(G_d(D''), \mathit{par}, G_e(E)) \\ &\Leftrightarrow \mathit{sinv}(G_d(D'), \mathit{par}, G_e(E)) \\ &\Leftrightarrow \mathit{sinv}(D', \mathit{par}, E) \end{aligned}$$

□

Lemma 32 (Safe Consistency Blocks Preserve Others' Simulation Relation) *Given (i) two consistency blocks σ and τ by some computation unit r which are (ii) running out of configurations D and E that (iii) are consistent wrt. to the sequential simulation relation for another unit $q \neq r$ using simulation parameter par. If (iv) σ and τ are safe wrt. given ownership annotations o and o' and if (v) the shared invariant holds for the resulting configurations D' and E', then also the simulation relation for q still holds.*

$$\begin{aligned} \forall D, E, \sigma, o, \tau, o', r, q, \mathit{par}. \quad & \textit{(i)} && \mathit{blk}(\sigma, r) \wedge \mathit{blk}(\tau, r) \wedge r \neq q \\ & \textit{(ii)} && D \overset{\langle \sigma, o \rangle}{\longmapsto} D' \wedge E \overset{\langle \tau, o' \rangle}{\longmapsto} E' \\ & \textit{(iii)} && \mathit{sim}_q(D.M, \mathit{par}, E.M) \\ & \textit{(iv)} && \mathit{safe}(D, \langle \sigma, o \rangle) \wedge \mathit{safe}(E, \langle \tau, o' \rangle) \\ & \textit{(v)} && \mathit{sinv}(D', \mathit{par}, E') \\ & \implies && \mathit{sim}_q(D'.M, \mathit{par}, E'.M) \end{aligned}$$

PROOF: Because the computations are safe, we can apply Lemma 20.1 inductively on all steps of $(D, \langle \sigma, o \rangle)$ and $(E, \langle \tau, o' \rangle)$ obtaining $D \approx_q D'$ and $E \approx_q E'$. Our claim then follows from Assumption 2. Note that in all applications p is instantiated with q. □

5.3.5 Simulation Proof

Now we can prove the concurrent simulation theorem.

PROOF OF THEOREM 6: by induction on $m = |\kappa|$. For $m = 0$ we set $\nu = \varepsilon$ to conclude claim (i) with $D' = D$ and $E' = E$. Claims (iii) and (iv) hold trivially and the second part of (ii) follows directly from hypothesis (v). Setting $o = \varepsilon$ as well, claim (v) collapses to $inv(D)$ and from hypothesis (iii) we get the safety of all complete consistency block machine computations running out of E. Therefore also $inv(E)$ must hold and we get $inv(D)$ from Corollary 2. We also get $P(E)$ from hypothesis (iii) which implies the well-formedness of all units in E by hypothesis (iv) on P and thus gives us the remaining first part of claim (ii).

As our induction hypothesis (IH) we assume the claim to hold for any consistency block machine schedule $\bar{\kappa}$ with arbitrary but fixed length $m-1$. Thus there exists a simulated complete consistency block machine computation $(E.M, \bar{\nu})$ such that all the desired properties already hold. Moreover $(D.M, \bar{\kappa})$ is safe wrt. ownership.

Taking the induction step $m-1 \to m$ we assume that we are given a block machine computation $(D.M, \kappa)$ with $|\kappa| = m$. We denote the last block by $\lambda = \kappa_m$ and the previous blocks by $\bar{\kappa} = \kappa[1:m)$, i.e., $\kappa = \bar{\kappa}\lambda$. From the induction hypothesis we get the $m-1$-step consistency block machine computation $(E.M, \bar{\nu})$ fulfilling the claims of the simulation theorem.

We set up ν accordingly for the first $m-1$ block steps, i.e., $\nu[1:m) = \bar{\nu}$ and introduce the intermediate machine states $\bar{M}_d$ and $\bar{M}_e$ (cf. Fig. 23). Then by induction hypothesis the following statements holds:

$$
\begin{array}{lll}
\textit{(a)} & \mathcal{CP}sched_c(E.M, \nu[1:m), par) \wedge D.M \overset{\kappa[1:m)}{\longmapsto} \bar{M}_d \wedge E.M \overset{\nu[1:m)}{\longmapsto} \bar{M}_e & \\
\textit{(b)} & \forall r.\ wf_r(\bar{M}_e) \wedge \forall r \in U_c(\bar{M}_d, par).\ wf_r(\bar{M}_d) & \\
\textit{(c)} & sim(\bar{M}_d, par, \bar{M}_e) & \\
\textit{(d)} & wb(D.M, \lfloor \kappa[1:m) \rfloor, par) \wedge \forall j < m.\ one\mathcal{IO}(\kappa_j, \nu_j) & \\
\textit{(e)} & \forall o_{\bar{\nu}}, \bar{\mathcal{G}}_e.\ E \overset{\langle \lfloor \nu[1:m) \rfloor, o_{\bar{\nu}} \rangle}{\longmapsto} (\bar{M}_e, \bar{\mathcal{G}}_e) \wedge safe(E, \langle \lfloor \nu[1:m) \rfloor, o_{\bar{\nu}} \rangle) & \Longrightarrow \\
\quad \exists o_{\bar{\kappa}}, \bar{\mathcal{G}}_d. & \textit{(e.1)} & D \overset{\langle \lfloor \kappa[1:m) \rfloor, o_{\bar{\kappa}} \rangle}{\longmapsto} (\bar{M}_d, \bar{\mathcal{G}}_d) \\
& \textit{(e.2)} & safe(D, \langle \lfloor \kappa[1:m) \rfloor, o_{\bar{\kappa}} \rangle) \\
& \textit{(e.3)} & sinv((\bar{M}_d, \bar{\mathcal{G}}_d), par, (\bar{M}_e, \bar{\mathcal{G}}_e))
\end{array}
$$

Thus we only need to take care of the last block λ which is non-empty by hypothesis and being executed by unit $p = \lambda_1.s$. Let d' be the final machine state of the concrete computation, i.e., $\bar{M}_d \overset{\lambda}{\longmapsto} d'$. Below we list the hypotheses for applying the sequential

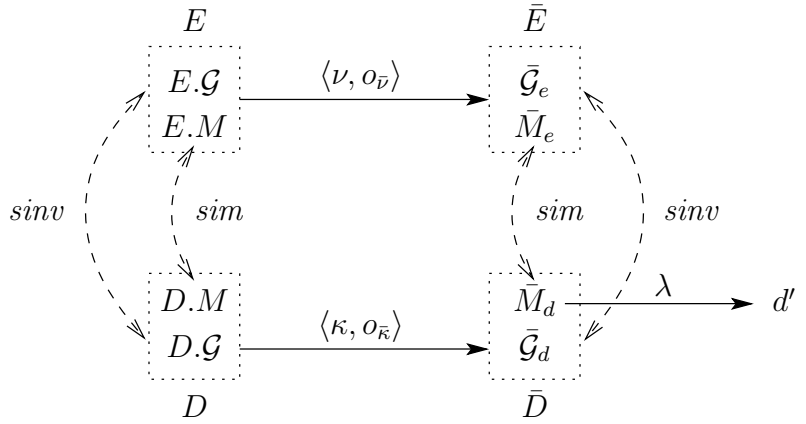

Figure 23: The induction hypothesis in the proof of the *Cosmos* model simulation theorem. Computation $(D.M, \bar{\kappa})$ is ownership-safe and simulates an abstract computation $(E.M, \bar{\nu})$.

simulation theorem on the possibly incomplete block $(\bar{M}_d, \lambda)$.

(i) $wf_p(\bar{M}_d) \wedge wf_p(\bar{M}_e)$

(ii) $sim_p(\bar{M}_d, par, \bar{M}_e) \wedge \mathcal{CP}_p(\bar{M}_d, par) \wedge \mathcal{CP}_p(\bar{M}_e, par)$

(iii) $blk(\lambda, p) \wedge suit(\lambda) \wedge \exists d'.\ \bar{M}_d \overset{\lambda}{\longmapsto} d'$

(iv) $\forall \pi, e'.\ \bar{M}_e \overset{\pi}{\longmapsto} e' \wedge blk(\pi, p) \wedge \mathcal{CP}_p(e', par) \implies sc(\bar{M}_e, \pi, par) \wedge wf_p(\bar{M}_e)$

Numbers (i) and (ii) follow directly from IH (b) and (a) on $\bar{M}_d$ and $\bar{M}_e$. Here we use that p is in a consistency point in $\bar{M}_d$ because κ is a consistency block machine schedule, where all blocks, including λ, start in consistency points. This also gives us $blk(\lambda, p)$ in claim (iii). We get $\mathcal{CP}_p(\bar{M}_e, par)$ because $\bar{\nu}$ is a complete consistency block schedule.

Moreover, we have $suit(\lambda)$ and that $(\bar{M}_d, \lambda)$ is a computation from hypothesis on κ. The last hypothesis follows from $safety_{cB}(E, P, par)$ which says that P holds in all consistency points reachable from E. Also by hypothesis (iv) P implies that any computation reaching such a point obeys the software conditions and leads into a well-formed unit state. Since $\bar{M}_e$ is reachable from E by IH, P also holds for any consistency point reached from there via some consistency block π. Then $(\bar{M}_e, \pi)$ obeys the software conditions and the resulting machine state is well-formed.

Thus we can apply the sequential simulation theorem for unit p on $d := \bar{M}_d$, $e := \bar{M}_e$, and $\omega := \lambda$. We obtain complete blocks σ and τ leading into machine states d'' and e'' which are consistency points for unit p.

$$\underset{oneI\mathcal{O}(\sigma,\tau)}{\phantom{\bar{M}_d \overset{\sigma}{\longmapsto} d''}} \quad \underset{blk(\tau,p)}{\bar{M}_d \overset{\sigma}{\longmapsto} d''} \quad \underset{\mathcal{CP}_p(d'', par)}{\bar{M}_e \overset{\tau}{\longmapsto} e''} \quad \underset{\mathcal{CP}_p(e'', par)}{\lambda \rhd_p^{blk} \sigma}$$

The simulation relation for p holds wrt. parameter par and the configurations of unit p

in d'' as well as e'' are well-formed .

$$sim_p(d'', par, e'') \qquad wf_p(d'') \qquad wf_p(e'')$$

Moreover the concrete computation $(\bar{D}.M, \sigma)$ is well-behaved and by hypotheses (iii) and (iv) computation $(\bar{E}.M, \tau)$ obeys the software conditions on computations of S_e.

$$wb(\bar{M}_d, \sigma, par) \qquad sc(\bar{M}_e, \tau, par)$$

Since $\nu[1:m)\tau$ forms a complete consistency block machine schedule, we know that $(E.M, \nu[1:m)\tau)$ is ownership-safe by hypothesis $safety_c(E, P, par)$ wrt. some ownership annotation $o_{\bar{\nu}}o_\tau$ with $|o_\tau| = |\tau|$, i.e.:

$$safe(E, \langle \lfloor \nu[1:m)\tau \rfloor, o_{\bar{\nu}}o_\tau \rangle)$$

Therefore, with $\bar{E} = (\bar{M}_e, \bar{\mathcal{G}}_e)$ and $E'' = (e'', \mathcal{G}''_e)$ for some ghost states $\bar{\mathcal{G}}_e$ and $\mathcal{G}''_e$, also $(\bar{E}, \langle \tau, o_\tau \rangle)$ obeys the ownership policy and the computation is leading into E''.

$$\bar{E} \overset{\langle \tau, o_\tau \rangle}{\longmapsto} E'' \qquad safe(\bar{E}, \langle \sigma, o_\tau \rangle)$$

In what follows we prove claim (v) assuming $o_\nu = o_{\bar{\nu}}o_\tau$ without loss of generality. From IH (e) we know there exists annotation $o_{\bar{\kappa}} \in \Omega^*_{S_d}$ and ghost state $\bar{\mathcal{G}}_d$ such that:

$$D \overset{\langle \lfloor \kappa[1:m) \rfloor, o_{\bar{\kappa}} \rangle}{\longmapsto} (\bar{M}_d, \bar{\mathcal{G}}_d) \ \wedge \ safe(D, \langle \lfloor \kappa[1:m) \rfloor, o_{\bar{\kappa}} \rangle) \ \wedge \ sinv((\bar{M}_d, \bar{\mathcal{G}}_d), par, (\bar{M}_e, \bar{\mathcal{G}}_e))$$

Using Assumption 1 we deduce that there exists $o_\sigma \in \Omega^*_{S_d}$ and $\mathcal{G}''_d$ such that:

$$(\bar{M}_d, \bar{\mathcal{G}}_d) \overset{\langle \sigma, o_\sigma \rangle}{\longmapsto} (d'', \bar{\mathcal{G}}''_d) \wedge safe((\bar{M}_d, \bar{\mathcal{G}}_d), \langle \sigma, o_\sigma \rangle) \wedge sinv((d'', \mathcal{G}''_d), par, E'')$$

This means that there exists an ownership annotation for σ such that the corresponding computation running out of $\bar{D} = (\bar{M}_d, \bar{\mathcal{G}}_d)$ is ownership-safe and leads into a configuration $D'' = (d'', \mathcal{G}''_d)$. Therefore the simulation relation for p holds also between D'' and E'' as well as the shared invariant.

$$sim_p(D''.M, par, E''.M) \qquad p \in U_c(D''.M, par) \qquad sinv(D'', par, E'')$$

By definition of $safe$, prefix λ of $\langle \sigma, \omega \rangle$ is also safe wrt. annotation $o_\lambda = o_\sigma[1:|\lambda|]$, i.e., $safe(\bar{D}, \langle \lambda, o_\lambda \rangle)$ holds and we set $o_\kappa = o_{\bar{\kappa}}o_\lambda$. Similarly, the concrete computation $(\bar{M}_d, \lambda)$ is well-behaved. Thus claims (v.a-b) and the first term of claim (iv) are proven using IH (d) and (e) as well as $D.M \overset{\kappa}{\longmapsto} d'$ and $M'_d = d'$:

$$\exists o_\kappa, \mathcal{G}'_d.\ D \overset{\langle \lfloor \kappa \rfloor, o \rangle}{\longmapsto} (M'_d, \mathcal{G}'_d) \ \wedge \ safe(D, \langle \lfloor \kappa \rfloor, o \rangle) \ \wedge \ wb(D.M, \lfloor \kappa \rfloor, par)$$

Now we perform a case split on whether λ contains an $\mathcal{IO}$ step and whether it is complete. We introduce sequence ω which extends λ forming σ, i.e., $\sigma = \lambda\omega$.

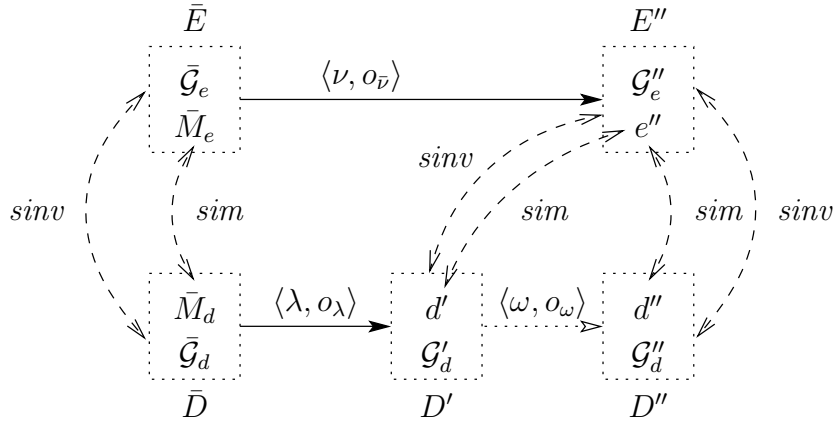

Figure 24: Case 2 in the induction step of the *Cosmos* model simulation proof. Here λ is incomplete but contains an $\mathcal{IO}$ step of unit p. Between d' and e'' relation sim holds, but sim_p does not.

1. $\omega = \varepsilon$ — In this case λ is already complete and we naturally include τ in ν as the last block that is simulated by λ, i.e., $\nu = \bar{\nu}\tau$. Most claims follow directly from previously shown statements by $D' = (M'_d, \mathcal{G}'_d) = D''$ and $E' = (M'_e, \mathcal{G}'_e) = E''$. Only the parts of claims (i), (ii), and (iii) for other units $q \neq p$ are missing.

   We see that a unit q is in a consistency point in D' iff it is in a consistency point in $\bar{D}$. This statement holds because the safe steps by p do not influence the local states of other units. Formally, by inductive application of Lemma 20.1 on the ownership-safe computation $(\bar{D}, \langle \lambda, o_\lambda \rangle)$ we obtain $\bar{D} \approx_{\overline{p}} D'$. This implies $\bar{D} \approx_q D'$ by definition and thus the unit states for all units q are equal in both configurations, i.e., $\bar{D}.M.u(q) = D'.M.u(q)$, and we conclude:

   $$\begin{aligned} q \in U_c(D'.M, par) &\iff \mathcal{CP}_q(D'.M, par) \iff \mathcal{CP}c(D'.M.u(q), par) \\ &\iff \mathcal{CP}c(\bar{D}.M.u(q), par) \iff q \in U_c(\bar{D}.M, par) \end{aligned}$$

   The same holds for units in E' and $\bar{E}$ similarly. In particular we then know that all units in E' are in consistency points.

   $$\forall q \in \mathbb{N}_{nu}.\ q \in U_c(M'_d, par)$$

   Thus we can directly conclude the remaining parts of claim (i) using IH (a), $blk(\tau, p)$, and $\mathcal{CP}_p(M'_e, par)$.

   $$\mathcal{CP}sched_c(E.M, \nu, par) \qquad E.M \overset{\nu}{\longmapsto} M'_e$$

   For all units $q \in U_c(\bar{D}.M, par)$ we now need to show that their simulation relation is not broken by $(\bar{D}, \langle \lambda, o_\lambda \rangle)$ and $(\bar{E}, \langle \tau, o_\tau \rangle)$. Since these computations are safe wrt. ownership we can apply Lemma 32 which gives us:

   $$\forall q \in U_c(M'_d, par).\ sim_q(M'_d, par, M'_e)$$

This is however the definition of $sim(M'_d, par, M'_e)$, thus claim (iii) holds.

For Claim (ii) we already have shown $wf_p(M'_e)$. For all other units $q \neq p$ we have $\bar{E} \approx_q E'$ by inductive application of Lemma 20.1 on $(\bar{E}, \langle \tau, o_\tau \rangle)$ as shown for $\bar{D}$ and D' above. The well-formedness in E' follows then from IH (b), the shared invariant on D' and E', and Assumption 3. Similarly, by $D \approx_q D'$, we have the well-formedness for all units of D' that are in consistency points.

$$\forall r \in \mathbb{N}_{nu}.\ wf_r(M'_e) \qquad \forall q \in U_c(D'.M, par).\ wf_q(M'_d)$$

Thus we have proven all claims for this case.

2. $\lambda|_{io} \neq \varepsilon$ — Consistency block λ contains already an $\mathcal{IO}$ operation (cf. Fig. 24). In this case we include τ in the simulated computation $(E.M, \nu)$ even if λ is incomplete, because we need to maintain the invariant on shared memory and ownership. We set $\nu = \bar{\nu}\tau$ and conclude the second part of claim (iv) using IH (d) and $one\mathcal{IO}(\sigma, \tau)$ which implies $one\mathcal{IO}(\lambda, \tau)$ because prefixes of sequences cannot contain more $\mathcal{IO}$ points than the original.

$$\forall j \leq m.\ one\mathcal{IO}(\kappa_j, \nu_j)$$

By $one\mathcal{IO}(\sigma, \tau)$ and $\lambda|_{io} \neq \varepsilon$ we know that ω does not contain any $\mathcal{IO}$ step, i.e., $\omega|_{io} = \varepsilon$. With the ownership annotation o_ω for the suffix, i.e., $o_\omega = o_\sigma(|\lambda| : |\omega|]$, it follows from the safety of $(\bar{D}, \langle \sigma, o_\sigma \rangle)$ where $(\bar{M}_d, \bar{\mathcal{G}}_d)$ that the corresponding computation from $D' = (d', \mathcal{G}'_d)$ to D'' is ownership-safe.

$$D' \overset{\langle \omega, o_\omega \rangle}{\longmapsto} D'' \qquad safe(D', \langle \omega, o_\omega \rangle)$$

Since we also have $sinv(D'', par, E'')$ we can apply Lemma 31 to obtain:

$$sinv(D', par, E'')$$

This proves claim (v.c) as we set $E' = (M'_e, \mathcal{G}'_e) = E''$. From $blk(\lambda\omega)$ we know that ω does not contain any interleaving-points. Furthermore, by $\mathcal{CP}sched(D.M, \kappa, par)$ we have $\mathcal{IPCP}(R_{S_d, S'_e}, par)$. Consequently p is not in a consistency point in D'.

$$\begin{aligned} \omega \neq \varepsilon \wedge blk(\lambda\omega) \quad &\Longrightarrow \quad /\omega_1.ip \quad \Longrightarrow \quad /\mathcal{IP}_p(D'.M, \omega.in) \\ &\Longrightarrow \quad /\mathcal{CP}_p(D'.M, par) \quad \Longrightarrow \quad p \notin U_c(D'.M, par) \end{aligned}$$

Therefore λ is incomplete and we do not need to show that the simulation relation holds for p between D' and E'. We show the remaining parts of claims (i), (ii), and (iii) like in the first case.

3. $\lambda|_{io} = \varepsilon \wedge \omega \neq \varepsilon$ — λ is an incomplete consistency block containing only local steps (cf. Fig. 25). In this case we cannot include τ in the simulated computation $(E.M, \nu)$ because it might contain $\mathcal{IO}$ steps that are introduced only in the extension ω. Such $\mathcal{IO}$ steps might modify the shared memory and ownership state, hence we could not prove the shared invariant between D' and E''.

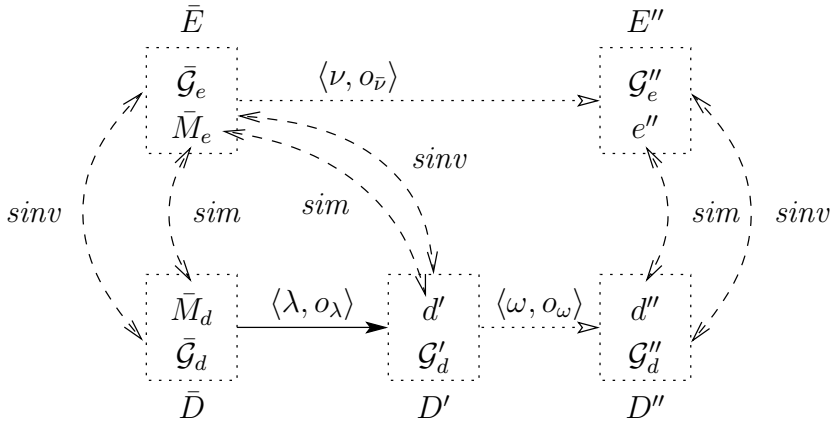

Figure 25: Case 3 in the induction step of the *Cosmos* model simulation proof. Here λ is incomplete and contains only local steps of unit p. However, ω might contain $\mathcal{IO}$ steps. Note that between d' and $\bar{M}_e$ sim holds, but not sim_p.

Therefore we omit the incomplete block in the simulation and set $E' = \bar{E}$, $\nu_m = \varepsilon$, as well as $o_\nu = o_{\bar{\nu}}$. Claim (i) then follows directly from IH (a). The same holds for the well-formedness of all units in E'. The well-formedness of units of D' is proven exactly the same way as in case 1. With this setting of ν and $\lambda|_{io} = \varepsilon$ we also get $one\mathcal{IO}(\kappa_m, \nu_m)$ and with IH (d) we complete the proof of claim (iv).

$$\forall j \leq m.\ one\mathcal{IO}(\kappa_j, \nu_j)$$

Again we use $safe(\bar{D}, \langle \lambda, o_\lambda \rangle)$ to show $\bar{D} \approx_q D'$ for all $q \neq p$. Applying Lemma 31 on $\bar{D}$, D', and $\bar{E}$ we obtain the shared invariant between D' and $E' = \bar{E}$.

$$sinv(D', par, E')$$

Similarly Lemma 32 with τ instantiated to be an empty sequence yields the simulation relation $sim_q(D'.M, par, E'.M)$ for all $q \neq p$ that are in consistency points as above. From $\omega \neq \varepsilon$ we know that unit p is not in a consistency point in D', hence we do not need to show that simulation holds for p. We obtain the final claim:

$$sim(D'.M, par, E'.M)$$

This finishes the overall *Cosmos* model simulation proof. □

Thus we have shown how to lift up sequential simulation theorems forming a system-wide concurrent simulation. If the simulation theorems hold locally and fulfill Assumptions 1 to 3, any interleaving of consistency blocks on the implementation level S_d will be consistent to some simulating complete block machine computation on the abstract level S_e. Moreover, in such a scenario, for every parallel program it suffices

to show memory safety on the abstract level as it can be transfered down, guaranteeing the safety of all well-behaved $\mathcal{IP}$ schedule computations of the implementation. Using the order reduction theorem safety can be transfered further down to arbitrarily interleaved schedules.

5.4 Applying the Order Reduction Theorem

Our order reduction theorem allows to transfer safety from safe $\mathcal{IP}$ schedules to arbitrarily interleaved *Cosmos* machine schedules. Remember that this safety transfer theorem has two hypotheses, namely that all $\mathcal{IP}$ schedule computations leaving configuration D are safe and fulfil the $\mathcal{IOIP}$ condition, saying that all units start in interleaving-points and that a unit always passes an interleaving-point between two $\mathcal{IO}$ steps. Now it would be desirable if we could use the *Cosmos* model simulation theorem proven above in order to obtain $safety_{\mathcal{IP}}(D, P)$ and $\mathcal{IOIP}_{\mathcal{IP}}(D)$. However we cannot prove these hypotheses of the order reduction theorem directly. Instead we can derive two weaker properties from the simulation theorem. With $\theta \in \Theta^*_{S_d}$, $o \in \Omega^*_{S_d}$, the predicates

$$\begin{aligned}
safety(D, P, suit) &\stackrel{def}{\equiv} \forall \theta.\ suit(\theta) \wedge comp(D.M, \theta) \implies \exists o.\ safe_P(D, \langle \theta, o \rangle) \\
safety_{\mathcal{IP}}(D, P, suit) &\stackrel{def}{\equiv} \forall \theta.\ \mathcal{IP}sched(\theta) \wedge suit(\theta) \wedge comp(D.M, \theta) \\
&\qquad\qquad \implies \exists o.\ safe_P(D, \langle \theta, o \rangle) \\
\mathcal{IOIP}_{\mathcal{IP}}(D, suit) &\stackrel{def}{\equiv} \forall \theta.\ \mathcal{IP}sched(\theta) \wedge suit(\theta) \wedge comp(D.M, \theta) \implies \mathcal{IOIP}(\theta)
\end{aligned}$$

denote the safety and $\mathcal{IOIP}$ condition for all ($\mathcal{IP}$) schedules that are suitable for simulation. We furthermore augment the machine states of S_d with a history variable wb similar to the sc flag of S'_e. The additional semantics for the extended machine S'_d with $d, d' \in \mathbb{M}_{S'_d}$, step $\alpha \in \Theta_{S_d}$, and parameter $par \in \mathcal{P}$ is given by:

$$d \overset{\alpha}{\mapsto} d' \implies d'.u(p).wb = d.u(p).wb \wedge wb(d, \alpha, par)$$

Now we define *Cosmos* machine safety property for a given parameter par that denotes good behaviour in the past (before D) for all computation units.

$$W : \mathbb{C}_{S_d} \to \mathbb{B} \qquad W(D) \stackrel{def}{\equiv} \forall p \in \mathbb{N}_{nu}.\ D.u_p.wb$$

Finally we define a shorthand for the simulation hypotheses.

Definition 82 (Simulation Hypotheses) *We define a predicate $simh$ to denote the hypotheses of the concurrent simulation theorem for a framework $(R_{S_d,S'_e}, sinv)$, start configurations $D \in \mathbb{C}_{S_d}$, $E \in \mathbb{C}_{S'_e}$, and a simulation parameter $par \in \mathcal{P}$. We demand that (i) all units D in consistency points are well-formed for all units, (ii) at least one unit is in a consistency point and for all units the wb flag is set to true, and (iii) consistent wrt. the simulation relation and shared invariant. We assume to have proven the sequential simulation theorem according to R_{S_d,S'_e}*

fulfilling the $\mathcal{IPCP}$ condition and Assumptions 1-3. Moreover (iv) memory safety is verified for all complete block computations starting in E along with a property P that (v) implies that computations running out of E obey the software conditions and preserve well-formedness.

$$\begin{aligned} simh(D,E,P,par) \equiv\ & \textit{(i)} && \forall p \in U_c(D.M, par).\ wf_p(D.M) \\ & \textit{(ii)} && \exists p \in \mathbb{N}_{nu}.\ \mathcal{CP}_p(D.M, par) \wedge W(D) \\ & \textit{(iii)} && sim(D.M, par, E.M) \wedge sinv(D, par, E) \\ & \textit{(iv)} && safety_{cB}(E, par, P) \wedge \mathcal{IPCP}(R_{S_d,S_e}, par) \\ & \textit{(v)} && \forall E' \in \mathbb{C}_{S'_e}, p \in \mathbb{N}_{nu}.\ P(E') \implies wf_p(E'.M) \wedge E'.u_p.sc \end{aligned}$$

5.4.1 Corollaries of the Simulation Theorem

Now we can prove the following corollaries of the concurrent simulation theorem.

Corollary 3 (Simulating $\mathcal{IP}$ Schedules are Safe and Well-Behaved) *Assuming a pair of simulating* Cosmos *machine configurations $D \in \mathbb{C}_{S'_d}$ and $E \in \mathbb{C}_{S'_e}$, if the simulation hypotheses hold for concurrent simulation framework $(R_{S'_d,S'_e}, sinv)$ and some parameter $par \in \mathcal{P}$, then all suitable $\mathcal{IP}$ schedule computations running out of D are ownership-safe and well-behaved.*

$$\forall D, E, par.\ simh(D, E, P, par) \implies safety_{\mathcal{IP}}(D, W, suit)$$

PROOF: We show that any computation $(D.M, \theta)$ with $\mathcal{IP}sched(\theta)$ and $suit(\theta)$ is well-behaved and safe wrt. ownership. To this end we apply Lemma 30 obtaining a block machine schedule κ with $\lfloor\kappa\rfloor = \theta$. Besides the hypotheses of $simh$ we have the following conditions on computation $(D.M, \lfloor\kappa\rfloor)$ for applying the simulation theorem.

$$\begin{aligned} &\textit{(i)} && \mathcal{CP}sched(\kappa, par) \\ &\textit{(ii)} && comp(D.M, \lfloor\kappa\rfloor) \\ &\textit{(iii)} && \forall \lambda \in \kappa.\ \lambda \neq \varepsilon \wedge suit(\lambda) \end{aligned}$$

We get (i) by definition because we have $\mathcal{IP}sched(\lfloor\kappa\rfloor)$ from hypothesis on $\theta = \lfloor\kappa\rfloor$ and $\mathcal{IPCP}(R_{S'_d,S'_e})$ from $simh$. Similarly (ii) follows from $comp(D.M, \theta)$. All steps in κ are suitable for simulation because of $suit(\theta)$. For all $\lambda \in \kappa$ we then have by definition of the suitability of sequences:

$$suit(\theta) \implies \forall \alpha \in \theta.\ suit(\alpha) \implies \forall \alpha \in \lfloor\kappa\rfloor.\ suit(\alpha) \implies \forall \alpha \in \lambda.\ suit(\alpha) \implies suit(\lambda)$$

Finally we can assume wlog. that κ does not contain empty blocks. If it did we can simply remove them without changing the semantics of the block machine computation. Thus we can apply Theorem 6 and obtain:

$$\exists o \in \Omega^*_{S'_d}.\ safe(D, \langle\theta, o\rangle) \qquad wb(D.M, \theta, par) \qquad D.M \overset{\langle\theta,o\rangle}{\longmapsto} D'$$

Here D' is the configuration machine state reached D by this annotated computation. By inductive application of the semantics of the wb flag on the well-behaved computation we get $W(D')$. Since we chose θ arbitrarily we have proven our claim. □

Corollary 4 ($\mathcal{IOIP}$ Assumption for Simulating $\mathcal{IP}$ Schedules) *Given a* Cosmos *machine simulation constrained as in the previous corollary with start configuration D, then every $\mathcal{IP}$ schedule that is suitable for simulation and running out of D fulfills the $\mathcal{IOIP}$ condition that any consistency block contains at most one $\mathcal{IO}$ operation and every unit starts in an interleaving-point.*

$$\forall D, E, par. \quad simh(D, E, P, par) \quad \Longrightarrow \quad \mathcal{IOIP}_{\mathcal{IP}}(D, suit)$$

PROOF: For a given suitable $\mathcal{IP}$ schedule θ, such that $comp(D.M, \theta)$ we obtain block machine computation $(D.M, \kappa)$ with $\lfloor\kappa\rfloor$ as above and apply the concurrent simulation theorem. From claim (iv) we get

$$\forall j \leq |\kappa|.\ one\mathcal{IO}(\kappa_j, \nu_j)$$

where ν is the abstract block schedule that is simulated by κ. From the definition of $one\mathcal{IO}$ we get $one\mathcal{IO}(\kappa_j) \vee \kappa_j = \varepsilon$ for all j and $\kappa_j = \varepsilon$ actually implies $one\mathcal{IO}(\kappa_j)$, i.e., all blocks of κ contain at most one $\mathcal{IO}$ step. In addition by $Bsched(\kappa)$ all blocks start in an interleaving-point and contain only steps by the same unit.

$$\forall j \leq |\kappa|.\ one\mathcal{IO}(\kappa_j) \wedge \exists p \in \mathbb{N}_{nu}.\ blk(\kappa_j, p)$$

We can rewrite the $\mathcal{IOIP}$ condition for transition sequence $\lfloor\kappa\rfloor$ as follows.

$$\begin{aligned}\mathcal{IOIP}(\lfloor\kappa\rfloor) \stackrel{def}{\equiv} \quad &(i) \quad \forall \sigma, \alpha, \tau, \beta, \omega, p.\ \lfloor\kappa\rfloor = \sigma\alpha\tau\beta\omega \wedge \alpha.io \wedge \beta.io \wedge \alpha.s = \beta.s = p \\ &\qquad\qquad \Longrightarrow \exists \gamma \in \tau\beta.\ \gamma.ip \wedge \gamma.s = p \\ &(ii) \quad \forall \tau, \alpha, \omega, p.\ \lfloor\kappa\rfloor = \tau\alpha\omega \wedge \alpha.s = p \wedge \tau|_p = \varepsilon \implies \alpha.ip\end{aligned}$$

Condition (ii) demands that any first step of a unit in $\lfloor\kappa\rfloor$ starts in an interleaving-point. For any transition α such that $\lfloor\kappa\rfloor = \tau\alpha\omega$ with $\alpha.s = p$ and $\tau|_p = \varepsilon$ we know by induction on the number of blocks in κ that there exists a block with index j in κ such that α is the first step of this block, i.e., $\alpha = \kappa_j[1]$, which implies $\alpha.ip$ by $blk(\kappa_j, p)$.

To prove the first condition we assume a partitioning of $\lfloor\kappa\rfloor$ as stated. For the $\mathcal{IO}$ steps α and β of unit p we know that there must exist blocks κ_i and κ_j with $i < j$ such that $\alpha \in \kappa_i$ and $\beta \in \kappa_j$. The steps cannot lie in the same block because we have $one\mathcal{IO}(\kappa_i)$ and $one\mathcal{IO}(\kappa_j)$. By $blk(\kappa_j, p)$ we know that $\kappa_j[1].ip$. Thus there exists a transition $\gamma = \kappa_j[1]$ in $\tau\beta$ such that $\gamma.ip$. This proves $\mathcal{IOIP}(\lfloor\kappa\rfloor)$ and thus $\mathcal{IOIP}(\theta)$. □

Thus we have shown that all computations starting in configuration D with $\mathcal{IP}$ schedules that are suitable for simulation are ownership-safe and fulfill the $\mathcal{IOIP}$ condition. However, according to the order reduction theorem we actually need ownership safety and the $\mathcal{IOIP}$ condition for *all* $\mathcal{IP}$ schedules running out of D. In order to bridge this gap we will take the following measures.

1. We prove that the suitability of schedules is preserved by reordering.
2. We extend the order reduction theorem to be applicable on the subset of suitable schedules.

This will allow us to derive the ownership-safety of arbitrary suitable schedules from the simulation hypotheses. Now we show the necessary lemmas step by step.

5.4.2 Suitability and Reordering

First we prove a useful property of the suitability predicate.

Lemma 33 (Suitability is Independent from Interleaving) *For R_{S_d,S'_e}, a given transition sequence θ is suitable for simulation iff its subsequences for each unit are.*

$$suit(\theta) \Longleftrightarrow \forall p \in \mathbb{N}_{nu}.\ suit(\theta|_p)$$

PROOF: We show the two directions of implication independently. By definition:

$$suit(\theta) \Longleftrightarrow \forall \alpha \in \theta.\ suit(\alpha)$$

Assuming for the sake of contradiction that $suit(\theta)$ holds and there is an $\alpha \in \theta|_p$ such that $/suit(\alpha)$, then by definition of the subsequence notation α is also contained in θ, i.e., $\exists \alpha \in \theta.\ /suit(\alpha)$, which contradicts our assumption $suit(\theta)$.

If $suit(\theta)$ does not hold then again there is an $\alpha \in \theta$ such that $/suit(\alpha)$. By definition we also have $\alpha \in \theta|_{\alpha.s}$ and since $\alpha.s \in \mathbb{N}_{nu}$ there exists a $p \in \mathbb{N}_{nu}$ such that $/suit(\theta|_p)$ holds with $p = \alpha.s$. □

Corollary 5 (Reordering Preserves Suitability) *Given are two equivalently reordered transition sequences θ and θ'. Then θ is suitable for simulation wrt. simulation framework R_{S_d,S'_e} iff θ' is suitable as well.*

$$\theta \mathrel{\hat{=}} \theta' \implies suit(\theta) \Leftrightarrow suit(\theta')$$

PROOF: By definition of the reordering relation we have $\theta|_p = \theta'_p$ (1) for all $p \in \mathbb{N}_{nu}$. Then with Lemma 33 we conclude:

$$suit(\theta) \overset{(L33)}{\Leftrightarrow} \forall p \in \mathbb{N}_{nu}.\ suit(\theta|_p) \overset{(1)}{\Leftrightarrow} \forall p \in \mathbb{N}_{nu}.\ suit(\theta'|_p) \overset{(L33)}{\Leftrightarrow} suit(\theta') \qquad \square$$

We combine this result with Lemma 10 and Lemma 11 obtaining a stronger Reordering Lemma. Also Lemma 24 can be strengthened.

Corollary 6 (Reordering Suitable Schedules into $\mathcal{IP}$ Schedules) *Given a transition sequence θ and simulation framework R_{S_d,S'_e}. If θ fulfills the $\mathcal{IOIP}$ condition and is suitable for simulation then we can find an equivalently reordered suitable $\mathcal{IP}$ schedule θ' which also fulfills the $\mathcal{IOIP}$ condition.*

$$\mathcal{IOIP}(\theta) \wedge suit(\theta) \implies \exists \theta'.\ \theta \mathrel{\hat{=}} \theta' \wedge \mathcal{IP}sched(\theta') \wedge \mathcal{IOIP}(\theta') \wedge suit(\theta')$$

PROOF: Using Lemma 11 we obtain θ' such that $\theta \mathrel{\hat{=}} \theta'$ and $\mathcal{IP}sched(\theta')$. By Lemma 10 we know that θ' fulfills the $\mathcal{IOIP}$ condition. Suitability follows from Corollary 5. □

Corollary 7 (Coverage for Suitable Computations) *Given a computation $(D.M, \theta)$ and simulation framework R_{S_d,S'_e} such that θ is suitable for simulation. From $safety_{\mathcal{IP}}(D, P, suit)$ and $\mathcal{IOIP}_{\mathcal{IP}}(D, suit)$ it follows that every schedule $\theta \in \Theta^*_{S_d}$ fulfills the $\mathcal{IOIP}$ condition and there exists an equivalently reordered suitable $\mathcal{IP}$ schedule computation.*

$$safety_{\mathcal{IP}}(D, P, suit) \wedge \mathcal{IOIP}_{\mathcal{IP}}(D, suit) \wedge comp(D.M, \theta) \wedge suit(\theta) \implies$$
$$\mathcal{IOIP}(\theta) \wedge \exists \theta'.\ \theta \mathrel{\hat{=}} \theta' \wedge \mathcal{IP}sched(\theta') \wedge comp(D.M, \theta') \wedge suit(\theta')$$

PROOF SKETCH: We simply replay the proof of Lemma 24 using the fact that by Corollary 5 all equivalently reordered schedules of θ are suitable for simulation. □

5.4.3 Order Reduction on Suitable Schedules

Finally we can show a stronger order reduction theorem that allows to transfer safety properties from the subset of suitable $\mathcal{IP}$ schedules down to suitable arbitrarily interleaved schedules.

Theorem 7 ($\mathcal{IP}$ Schedule Order Reduction for Suitable Schedules) *Given a simulation framework $R_{S'_d,S'_e}$ and a* Cosmos *model configuration $D \in \mathbb{C}_{S'_d}$ for which it has been verified that all suitable $\mathcal{IP}$ schedules originating in D are safe wrt. ownership and a* Cosmos *machine property P. Moreover all suitable $\mathcal{IP}$ schedule computations running out of D obey the $\mathcal{IOIP}$ condition. Then ownership safety and P hold on all computations with a schedule suitable for simulation that starts in D.*

$$safety_{\mathcal{IP}}(D, P, suit) \wedge \mathcal{IOIP}_{\mathcal{IP}}(D, suit) \implies safety(D, P, suit)$$

PROOF: Given a *Cosmos* machine computation $(D.M, \theta)$ with $suit(\theta)$ we obtain an equivalent, suitable, interleaving-point schedule computation $(D.M, \theta')$ by Corollary 7, i.e., $\theta \hat{=} \theta'$, $\mathcal{IP}sched(\theta')$, $comp(D.M, \theta')$, and $suit(\theta')$ holds. Then by $safety_{\mathcal{IP}}$ there exists an o' such that $(D, \langle \theta', o' \rangle)$ is safe. We conclude $\exists o.\ safe_P(D, \langle \theta, o \rangle)$ just like in the proof of Theorem 1. □

Note that for a trivial instantiation of $suit(\alpha) \equiv 1$, the new order reduction theorem implies the old one. Furthermore we can now wrap up the ownership transfer from abstract block machine schedules down to suitable arbitrarily interleaved concrete schedules as a corollary.

Corollary 8 (Ownership Transfer for Simulating Block Machines) *Assuming a pair of simulating* Cosmos *machine configurations $D \in \mathbb{C}_{S'_d}$ and $E \in \mathbb{C}_{S'_e}$, if the simulation hypotheses hold for concurrent simulation framework $(R_{S'_d,S'_e}, sinv)$ and some parameter $par \in \mathcal{P}$, then all suitable computations running out of D are ownership-safe and well-behaved.*

$$\forall D, E, par.\quad simh(D, E, P, par) \implies safety(D, W, suit)$$

PROOF: By Corollaries 3 and 4 we get:

$$safety_{\mathcal{IP}}(D, W, suit) \qquad \mathcal{IOIP}_{\mathcal{IP}}(D, suit)$$

Applying Theorem 7 we obtain $safety(D, W, suit)$ □

Thus we have proven that the assumptions on the sequential simulation theorem, in particular Assumption 1, suffice to establish the transfer of ownership-safety from the abstract to the concrete concurrent *Cosmos* machine. In the next section we will consider simulation between complete block machine computations and treat the transfer of arbitrary *Cosmos* machine safety properties between simulation layers.

5.5 Property Transfer and Complete Block Simulation

Above we have shown the existence of a simulation between any concrete consistency block machine computation and a complete abstract block machine computation. Moreover we have proven property transfer for memory safety. For the transfer of other safety properties it is important to remember how the simulation proof was conducted.

The sequential simulation was proven to hold only for units that are in consistency points in the computation of the concrete *Cosmos* machine. For all other units no statement could be made about their states and locally owned memory regions. However the shared invariant on shared memory and the ownership state was proven to hold in all configurations of a simulating computation.

This has influence on the kind of properties we can transfer from the abstract down to the concrete simulation level. We will have to distinguish between global and local properties. Moreover safety properties proven on the abstract level do not translate one-to-one to the concrete level because we are dealing with different *Cosmos* machine instantiations. The "translation" of the verified abstract safety properties to properties of the concrete machine is achieved via the coupling relations between configurations of S'_d and S'_e, i.e., by the shared invariant for global properties, and by the sequential simulation relation for local properties of units in consistency points.

This notion of *simulated Cosmos machine properties* is formalized below. We finish the section by proving transfer of *Cosmos* machine safety properties for complete and incomplete block machine schedules.

5.5.1 Simulated *Cosmos* machine Properties

As explained above we cannot transfer a verified *Cosmos* machine property P from the abstract to the concrete simulation level. Naturally P is formulated in terms of S'_e and we cannot apply it to configurations of S'_d. However we can translate P into a *simulated Cosmos machine property* $\hat{Q}$ which holds for $D \in \mathbb{C}_{S'_d}$ iff P holds in a completely consistent state $E \in \mathbb{C}_{S'_e}$. Here we follow the approach of Cohen and Lamport for property transfer [CL98]. Nevertheless we cannot translate arbitrary properties. First, they must be *divisible* in global and local sub-properties.

Definition 83 (Divisible *Cosmos* machine Safety Property) *We say that P is a divisible* Cosmos *machine safety property on the abstract machine S'_e iff it has the following structure*

$$\forall E \in \mathbb{C}_{S'_e}.\ P(E) \equiv P_g(E) \wedge \forall p.\ P_l(E, p)$$

where P_g is a global property which depends only on shared resources and the ownership model and P_l constitutes local properties for each unit of the system. Consequently they are constrained as shown below for any $E, E' \in \mathbb{C}_{S'_e}$.

$$E \overset{s}{\sim} E' \wedge E \overset{o}{\sim} E' \implies P_g(E) = P_g(E')$$

$$\forall p.\ E \approx_p E' \implies P_l(E, p) = P_l(E', p)$$

The distinction of global and local properties is motivated by the simulation proof. Global properties are only restricting the shared memory and ownership state, the part of the configuration that is covered by the shared invariant which is holding at all times between simulating computations. Conversely, local properties depend on the local configuration of a single unit, which are only coupled with the implementation at consistency points. Thus we can translate global properties in all intermediate configurations using the shared invariant and translate local properties in consistency-points using the simulation relation.

Arbitrary safety properties that couple shared memory with local data, or couple the local data of several units, can in general not be translated because the involved computation units might never be in consistency-points at the same time. Technically we forbid safety properties that are stated as a disjunction of global and local properties. However this is not a crucial restriction and we could without problems allow properties of the form $P_g(C) \vee P_l(C)$ if needed. The notion of the property translation is formalized as follows.

Definition 84 (Simulated *Cosmos* machine Property) *Let P be a divisible* Cosmos *machine safety property on $\mathbb{C}_{S'_e}$ and $(R_{S'_d,S'_e}, sinv)$ be a concurrent simulation framework between machines S'_e and S'_d. Then for a given simulation parameter $par \in \mathcal{P}$ the simulated* Cosmos *machine property $\hat{Q}[P, par] : \mathbb{C}_{S'_d} \to \mathbb{B}$ can be derived by solving the following formula, which states for any configuration $E \in \mathbb{C}_{S'_e}$ being completely consistent with $D \in \mathbb{C}_{S'_d}$ that $\hat{Q}[P, par]$ holds in D iff P holds in E.*

$$\forall D, E.\ sinv(D, par, E) \wedge \forall p.\ sim_p(D.M, par, E.M) \implies (\hat{Q}[P, par](D) = P(E))$$

Note that $\hat{Q}[P, par]$ may be undefined for certain properties P.[2] *Moreover, as $\hat{Q}[P, par]$ should be a divisible* Cosmos *machine property, we must be able to split it into global part $\hat{Q}[P, par]_g$ and local parts $\hat{Q}[P, par]_l$ such that:*

$$\hat{Q}[P, par](D) = \hat{Q}[P, par]_g(D) \wedge \forall p.\ \hat{Q}[P, par]_l(D, p)$$

Consequently the following constraints must hold for $\hat{Q}[P, par]$.

$$\begin{aligned} \forall D, E. \quad & P_g(E) \wedge sinv(D, par, E) \implies \hat{Q}[P, par]_g(D) \\ \forall D, E, p. \quad & P_l(E, p) \wedge sim_p(D.M, par, E.M) \implies \hat{Q}[P, par]_l(D, p) \end{aligned}$$

While it is desireable to have local properties hold for all units, we have seen that for configurations in incomplete consistency block machine computations there are units for which the sequential simulation and thus local simulated properties do not hold. Therefore we have to relax the definition of simulated properties and introduce *incompletely simulated Cosmos machine properties*.

[2] This can be the case when P argues about components of S'_e that are not coupled with the concrete level S'_d via the simulation relation and shared invariant. Typically ghost state components fall into this category if they do not have counterparts in the ghost state of the implementation.

Definition 85 (Incompletely Simulated Cosmos Machine Property) *For a given* Cosmos *machine property P, concurrent simulation framework $(R_{S'_d,S'_e}, sinv)$, simulation parameter $par \in \mathcal{P}$, and configurations $D \in \mathbb{C}_{S'_d}, E \in \mathbb{C}_{S'_e}$ we define an incompletely simulated* Cosmos *machine property $Q[P, par] : \mathbb{C}_d \to \mathbb{B}$ below.*

$$Q[P, par](D) \equiv \hat{Q}[P, par]_g(D) \wedge \forall p \in U_c(D.M, par).\ \hat{Q}[P, par]_l(D, p)$$

Its definition uses the global and local parts of the simulated *Cosmos* machine property $\hat{Q}[P, par]$. The global part should hold for all configurations in a block schedule D and the local properies only if the corresponding machine is in a consistency point.

5.5.2 Property Transfer

Finally we prove the transfer of safety properties from the abstract simulation level down to arbitrary consistency block schedules on the concrete level.

Theorem 8 (Simulated Safety Property Transfer) *Given are a concurrent simulation framework consistent $(R_{S'_d,S'_e}, sinv)$ with $par \in \mathcal{P}$ and start configurations $D \in \mathbb{C}_{S'_d}, E \in \mathbb{C}_{S'_e}$ such that the simulation hypotheses are fulfilled. In particular if we have verified ownership-safety and a* Cosmos *machine property P for all complete block machine computations starting in E and P translates into the incompletely simulated* Cosmos *machine property $Q[P, par]$, then any suitable* Cosmos *machine schedule leaving D is safe wrt. ownership, $Q[P, par]$ holds for all reachable configurations, and all implementing computations are well-behaved.*

$$simh(D, E, P, par) \implies safety(D, Q[P, par] \wedge W, suit)$$

PROOF: We assume any $\mathcal{IP}$ schedule computation $(D.M, \theta)$ where θ is suitable for simulation and $D.M \overset{\theta}{\longmapsto} M'$ for some final machine state M'. By Lemma 30 we obtain block machine schedule κ such that $\lfloor \kappa \rfloor = \theta$. Since the hypotheses of Theorem 6 are fulfilled we can apply it to $(D.M, \kappa)$ and we get that there exists an ownership annotation o, abstract annotated schedule $\langle \lfloor \nu \rfloor, o' \rangle$ and final configurations D', E' such that $(D, \langle \theta, o \rangle)$ is a *Cosmos* machine computation leading also into machine state M', $(D.M, \theta)$ is well-behaved, and $(E.M, \nu)$ is a complete consistency block machine compution. Moreover the shared invariant as well as the sequential simulation relation for units in consistency points holds between D' and E'.

$$\begin{array}{c} safe(D, \langle \theta, o \rangle) \qquad wb(D.M, \theta) \qquad D \overset{\langle \theta, o \rangle}{\longmapsto} D' \qquad D'.M = M' \\ \mathcal{CP}sched_c(E, \nu, par) \qquad E \overset{\langle \lfloor \nu \rfloor, o' \rangle}{\longmapsto} E' \\ sinv(D', par, E') \qquad \forall p \in U_c(D', par).\ sim_p(M', par, E'.M) \end{array}$$

Now by hypothesis $safety_{cB}(E, P, par)$ we have $P(E')$ and thus by definition also the simulated property $Q[P, par](D')$ holds. As $(D.M, \theta)$ is well-behaved we also have $W(D')$ using the semantics of the wb flag. Since θ was choosen arbitrarily, we deduce

the following statement.

$$\forall \theta \in \Theta^*_{S'_d}.\, \mathcal{IP}sched(\theta) \wedge suit(\theta) \wedge comp(D.M, \theta) \implies \exists o \in \Omega^*_{S'_d}.\, safe_{Q[P,par]\wedge W}(D, \langle \theta, o \rangle)$$

This is however the definition of $safety_{\mathcal{IP}}(D, Q[P, par] \wedge W, suit)$ and Corollary 4 yields $\mathcal{IOIP}_{\mathcal{IP}}(D, suit)$. By Theorem 7 we conclude $safety(D, Q[P, par] \wedge W, suit)$. □

Thus the incompletely simulated *Cosmos* machine property for any P is maintained on the concrete level by the concurrent simulation. We can easily show that the simulated properties hold completely on complete consistency block machine computations. This is usefull for constructing pervasive concurrent model stacks where the safety of complete block machine computations is needed on the concrete level, which becomes the abstract level of the underlying simulation layer.

Corollary 9 (Complete Simulated Property Transfer) *Given are a concurrent simulation framework consistent* $(R_{S'_d,S'_e}, sinv)$ *with* $par \in \mathcal{P}$ *and start configurations* $D \in \mathbb{C}_{S'_d}$, $E \in \mathbb{C}_{S'_e}$ *such that the simulation hypotheses are fulfilled and a* Cosmos *machine property* P *is verified for all complete consistency block machines of* S'_e*. Then, if* $\hat{Q}[P, par]$ *exists, it holds for all complete consistency block machine computations of machine* S'_d*.*

$$simh(D, E, P, par) \implies safety_{cB}(D, \hat{Q}[P, par] \wedge W, suit)$$

PROOF: Given a block machine computation $(D.M, \kappa)$ that is complete and leading into computation M' where all units are in consistency points , i.e.:

$$\mathcal{CP}sched_c(D.M, \kappa, par) \qquad D.M \overset{\kappa}{\longmapsto} M' \qquad \forall p \in \mathbb{N}_{nu}.\, \mathcal{CP}_p(M', par)$$

By definition we have $Bsched(\kappa)$ and by Lemma 29 $\lfloor \kappa \rfloor$ is an $\mathcal{IP}$ schedule and the computation $(D.M, \lfloor \kappa \rfloor)$ leads also into M' by definition of the step sequence notation for block machine schedules.

$$\mathcal{IP}sched(\lfloor \kappa \rfloor) \qquad D.M \overset{\lfloor \kappa \rfloor}{\longmapsto} M'$$

Nevertheless by Theorem 8 we know that all *Cosmos* machine computations running out of D are safe wrt. ownership and property $Q[P, par] \wedge W$. Therefore we have:

$$\exists o, D'.\, safe(D, \langle \lfloor \kappa \rfloor, o \rangle) \wedge D \overset{\langle \lfloor \kappa \rfloor, o \rangle}{\longmapsto} D' \wedge D'.M = M' \wedge Q[P, par](D') \wedge W(D')$$

In particular the definition of $Q[P, par]$ gives us:

$$\hat{Q}[P, par]_g(D') \qquad \forall p \in U_c(M', par).\, \hat{Q}[P, par]_l(D', p)$$

Since all machines in M' are in consistency points we immediately get $\hat{Q}[P, par](D')$ by definition and $safe_{\hat{Q}[P,par]\wedge W}(D, \langle \lfloor \kappa \rfloor, o \rangle)$ follows. Our claim holds by definition of $safety_{cB}$ because we have chosen block machine schedule κ arbitrarily. □

This finishes our simulation theory. We have shown how to conduct simulation and transfer properties between abstract and concrete *Cosmos* models. However in order the illustrate the usability of our framework for reordering and simulation we will return to our *Cosmos* model instantiations below and establish the concurrent simulation theorems between MIPS and C-IL, or MASM respectively.

5.6 Instantiations

In the previous chapters we have introduced the *Cosmos* machines S^n_{MASM} and $S^n_{\text{C-IL}}$ which where instantiated according to the MASM and C-IL semantics presented earlier. We also defined sequential consistency relations and correctness theorems for the assembling or compilation of MASM and C-IL resulting in programs running on the MIPS ISA level. In the remainder of this chapter we will revisit the sequential correctness theorems and instantiate our concurrent simulation framework accordingly. Thus we will justify the concurrent MASM and C-IL semantics by sketching a simulation between S^n_{MIPS} and S^n_{MASM}, as well as S^n_{MIPS} and $S^n_{\text{C-IL}}$. Note that for both simulations we set parameter A_{code} of the MIPS machine equal to $\{a \in \mathcal{A} \mid \langle a \rangle \in CR\}$.

5.6.1 Concurrent MASM Assembler Consistency

Before we instantiate the concurrent simulation theorem let us revisit the MASM *Cosmos* machine S^n_{MASM} and the sequential MASM assembler correctness theorem presented in Section 4.2.2.

Sequential Simulation Framework

We have already given a definition of the sequential MIPS-MASM simulation framework $R_{S^n_{\mathrm{MIPS}},S^n_{\mathrm{MASM}}}$ before. We reproduce it below.

$$R_{S^n_{\mathrm{MIPS}},S^n_{\mathrm{MASM}}}.\begin{cases}\mathcal{P} & = & InfoT_{\mathrm{MASM}}\\ sim(h, info_\mu, l) & = & consis_{\mathrm{MASM}}((l.c, \lceil l.m \rceil), info_\mu, h)\\ \mathcal{CP}a(c, info_\mu) & = & 1\\ \mathcal{CP}c(c, info_\mu) & = & (c.pc \in A^{\mathrm{MASM}}_{cp})\\ wfa(l) & = & wf_{\mathrm{MASM}}(l.c, \lceil l.m \rceil)\\ sc(M, t, info_\mu) & = & sc_{\mathrm{MASM}}((M.u(t.s), \lceil M.m \rceil), info_\mu)\\ wfc(h) & = & wf^{\mathrm{MASM}}_{\mathrm{MIPS}}(h)\\ suit(\alpha) & = & suit^{\mathrm{MASM}}_{\mathrm{MIPS}}(\alpha.in)\\ wb(M, t, info_\mu) & = & wb^{\mathrm{MASM}}_{\mathrm{MIPS}}((M.u(t.s), M.m), t.in)\end{cases}$$

Here A^{MASM}_{cp} represents the instruction addresses of all consistency-points on the MIPS level and was defined as follows.

$$A^{\mathrm{MASM}}_{cp} \equiv \{adr(info_\mu, p, loc) \mid p \in \mathrm{dom}(\pi) \wedge loc \leq |\pi(p).body|\}$$

Note that this setting of the consistency points implies $\mathcal{IPCP}(R_{S^n_{\mathrm{MIPS}},S^n_{\mathrm{MASM}}}, info_\mu)$. The definition of $\mathcal{IO}$ steps on the MIPS level was given as follows.

$$S^n_{\mathrm{MIPS}}.\mathcal{IO}(u,m,eev) \equiv /jisr((u,\lceil m\rceil),eev) \wedge u.pc \in A_{io}$$

Here set A_{io} denotes the set of addresses of instructions that perform $\mathcal{IO}$ operations. Note that such a definition assumes that instructions are only fetched from the code region that is fixed by MASM code consistency (see $codeinv$). Considering $\mathcal{IO}$ steps in MASM, such steps can either be Compare-and-Swap instructions or memory operations that target global shared variables using to the global variables pointer as source register and offsets from the set off_{gs}. Depending on MASM program π we define a predicate to detect whether statement j of procedure p is an $\mathcal{IO}$ instruction.

$$io(\pi,p,j) \stackrel{def}{\equiv} \mathrm{cas}(\pi(p).body[j]) \vee gsv(\pi(p).body[j])$$

Then A_{io} is defined as follows.

$$A_{io} \stackrel{def}{\equiv} \{adr(info_\mu,p,loc) \mid p \in \mathrm{dom}(\pi), loc \leq |\pi(p).body|.\ io(\pi,p,loc)\}$$

If we take the generalized sequential simulation theorem literally then we need to prove the following statement for the *Cosmos* machines S^n_{MIPS} and S^n_{MASM}. Let $M_h \in \mathbb{M}_{S^n_{\mathrm{MIPS}}}$ and $M_\mu \in \mathbb{M}_{S^n_{\mathrm{MASM}}}$ in:

$$\begin{aligned}
&\forall M_h, M_\mu, info_\mu, \omega, p.\\
&\quad \textit{(i)} \quad wf^{\mathrm{MASM}}_{\mathrm{MIPS}}(M_h.u_p, M_h.m) \wedge wf_{\mathrm{MASM}}(M_\mu.u_p, \lceil M_\mu.m\rceil)\\
&\quad \textit{(ii)} \quad consis_{\mathrm{MASM}}((M_\mu.u_p, \lceil M_\mu.m\rceil), info_\mu, (M_h.u_p, M_h.m)) \wedge M_h.u_p.pc \in A^{\mathrm{MASM}}_{cp}\\
&\quad \textit{(iii)} \quad blk(\omega,p) \wedge \forall \alpha \in \omega.\ suit^{\mathrm{MASM}}_{\mathrm{MIPS}}(\alpha.in) \wedge \exists M''_h.\ M_h \stackrel{\omega}{\longmapsto} M''_h\\
&\quad \textit{(iv)} \quad \forall \pi\alpha, M''_\mu, M'''_\mu.\ M_\mu \stackrel{\pi}{\longmapsto} M''_\mu \stackrel{\alpha}{\mapsto} M'''_\mu \wedge blk(\pi\alpha,p)\\
&\qquad\qquad \Longrightarrow\ sc_{\mathrm{MASM}}((M''_\mu.u_p, \lceil M''_\mu.m\rceil), info_\mu) \wedge wf_{\mathrm{MASM}}(M'''_\mu.u_p, \lceil M'''_\mu.m\rceil)\\
\Longrightarrow\ &\exists \sigma, \tau, M'_h, M'_\mu.\\
&\quad \textit{(i)} \quad \omega \rhd^{blk}_p \sigma \wedge blk(\tau,p) \wedge one\mathcal{IO}(\sigma,\tau)\\
&\quad \textit{(ii)} \quad M_h \stackrel{\sigma}{\longmapsto} M'_h \wedge wf_{\mathrm{MIPS}}(M'_h.u_p, M'_h.m) \wedge\\
&\qquad\qquad \forall \theta\alpha\theta', M''_h.\ \sigma = \theta\alpha\theta' \wedge M_h \stackrel{\theta}{\longmapsto} M''_h \ \Longrightarrow\ wb^{\mathrm{MASM}}_{\mathrm{MIPS}}((M''_h.u_p, M''_h.m), \alpha.in)\\
&\quad \textit{(iii)} \quad M_\mu \stackrel{\tau}{\longmapsto} M'_\mu \wedge wf_p(M'_\mu.u_p, M'_\mu.p.m)\\
&\quad \textit{(iv)} \quad consis_{\mathrm{MASM}}((M'_\mu.u_p, \lceil M'_\mu.m\rceil), info_\mu, (M'_h.u_p, M'_h.m)) \wedge M'_h.u_p.pc \in A^{\mathrm{MASM}}_{cp}
\end{aligned}$$

However in this formulation the *Cosmos* model formalism and our step notation seem to obstruct hypotheses and claims of the theorem. In order to illustrate the actual sequential simulation that we have to prove for MASM, we fill in the definitions from the MASM consistency section and restate the theorem focussing on the local configuration of unit p. Deriving the theorem above from the one given below is merely a technicality.

Theorem 9 (Sequential MIPS-MASM Simulation Theorem) *Given a MASM starting configuration $c_{\mu_0} \in \mathbb{C}_{\mathrm{MASM}}$ that is (i) well-formed, a MIPS configuration $h_0 \in \mathbb{H}_{\mathrm{MIPS}}$ that has external device interrupts disabled and is (ii) consistent to c_{μ_0} wrt. some $info_\mu \in InfoT_{\mathrm{MASM}}$, and an external event vector sequence $eev \in (\mathbb{B}^{256})^*$, that (iii) contains no active reset signals. If (iv) every computation running out of c_{μ_0} leads into a well-formed state, that does not produce runtime-errors or stack overflows, and from where the next step does not access the stack or code memory regions,*

$$
\begin{aligned}
\forall c_{\mu_0}, h_0, eev, info_\mu \,.\quad & \textit{(i)} && wf_{\mathrm{MASM}}(c_{\mu_0}) \wedge h_0.c.spr[\mathrm{dev}] = 0 \\
& \textit{(ii)} && consis_{\mathrm{MASM}}(c_{\mu_0}, info_\mu, h_0) \wedge h_0.c.pc \in A^{\mathrm{MASM}}_{cp} \\
& \textit{(iii)} && \forall v \in \mathbb{B}^{256}.v \in eev \implies v[0] = 0 \\
& \textit{(iv)} && \forall c'_\mu \in \mathbb{C}_{\mathrm{MASM}}.\, c_{\mu_0} \longrightarrow^*_{\delta_{\mathrm{MASM}}} c'_\mu \implies \\
& && \qquad wf_{\mathrm{MASM}}(c'_\mu) \wedge \delta(c'_\mu) \neq \bot \\
& && \qquad \wedge\, /stackovf(c'_\mu, info_\mu) \wedge msp_\mu \geq 0 \\
& && \qquad \wedge\, /badmemop(c_\mu, info_\mu)
\end{aligned}
$$

then there exists an ISA computation with n steps that (i) is starting in h_0 and is computing according to external event vector sequence eev extended with another input sequence eev'. The computation leads into a state where device interrupts are disabled and that is (ii) consistent with the MASM state obtained by stepping c_{μ_0} once. Moreover (iii) that next MASM state is well-formed and (iv) the implementing ISA computation does not produce any interrupts nor is it fetching from addresses outside the code region. Finally (v)+(vi) the implementing computation contains exactly the same number of $\mathcal{IO}$ instructions as the implemented step of the MASM machine and at most one such operation.

$$
\begin{aligned}
\implies \quad & \exists n \in \mathbb{N}_0, h \in \mathbb{H}^{n+1}, eev' \in (\mathbb{B}^{256})^{n-|eev|}. \\
& \textit{(i)} && h_1 = h_0 \wedge h_1 \longrightarrow^n_{\delta_{\mathrm{MIPS}}, eev \circ eev'} h_{n+1} \wedge h_{n+1}.c.spr[dev] = 0 \\
& \textit{(ii)} && consis_{\mathrm{MASM}}(\delta_{\mathrm{MASM}}(c_{\mu_0}), info_\mu, h_{n+1}) \wedge h_{n+1}.c.pc \in A^{\mathrm{MASM}}_{cp} \\
& \textit{(iii)} && wf_{\mathrm{MASM}}(\delta_{\mathrm{MASM}}(c_{\mu_0})) \\
& \textit{(iv)} && \forall i \in \mathbb{N}_n.\, /jisr(h_i.c, I(h_i), eev_i) \wedge [\langle h_i.c.pc \rangle : \langle h_i.c.pc \rangle + 3] \subseteq CR \\
& \textit{(v)} && (\exists i \in \mathbb{N}_n.\, h_i.c.pc \in A_{io}) \iff \mathrm{cas}(I(c_\mu)) \vee gsv(I(c_\mu)) \\
& \textit{(vi)} && \forall i, j \in \mathbb{N}_n.\, h_i.c.pc \in A_{io} \wedge h_j.c.pc \in A_{io} \implies i = j
\end{aligned}
$$

After proving this theorem we can easily establish the former statement by constructing the simulating sequence σ using $eev \circ eev'$ for the inputs. The io flag in σ and τ is only set for steps that are performing $\mathcal{IO}$ operations according to the program counter and A_{io}. Since both computations agree on whether they contain an $\mathcal{IO}$ step and there is at most one such step, we have $one\mathcal{IO}(\sigma, \tau)$. Similarly, the ip flag is set only for the first step in σ and τ. The start configurations for σ and τ are in consistency points and lacking control consistency there cannot be further consistency points in the implementation of the single MASM step. Then with the $\mathcal{IPCP}$ condition we know that h_0 and c_{μ_0} are also in interleaving-points and σ, τ are consistency blocks.

Shared Invariant and Concurrent Simulation Assumptions

For establishing a concurrent simulation between MIPS and MASM we first of all need to define invariant on shared memory and the ownership state. In general we demand that the shared memory as well as the ownership configuration is identical. Nevertheless we need to take into account that for MASM the code and stack region is excluded from the memory address range. On the ISA level the stack region dedicated to unit p is always owned by p. By construction the code region lies in the set of read-only addresses and the read-only set is empty for S^n_{MASM}.

Definition 86 (Shared Invariant for Concurrent MIPS-MASM Simulation) *Given memories m_h, m_μ, read-only sets $\mathcal{R}_h$, $\mathcal{R}_\mu$, sets of shared addresses $\mathcal{S}_h$ and $\mathcal{S}_\mu$, as well as ownership mappings $\mathcal{O}_h$ and $\mathcal{O}_\mu$, we define the shared invariant for concurrent simulation of S^n_{MASM} by S^n_{MIPS} wrt. assembler information $info_\mu$ as follows. We demand (i) that memory contents are equal for all but the stack and code regions, that (ii) the shared addresses are equal, and (iii) that all units own the same addresses on the MIPS level as on the MASM level plus the individual stack region.*

$$
\begin{aligned}
&sinv^{\mathrm{MASM}}_{\mathrm{MIPS}}((m_h, \mathcal{S}_h, \mathcal{R}_h, \mathcal{O}_h), info_\mu, (m_\mu, \mathcal{S}_\mu, \mathcal{R}_\mu, \mathcal{O}_\mu)) \equiv \\
&\quad \textit{(i)} \quad m_h|_{S^n_{\mathrm{MASM}}.\mathcal{A}} = m_\mu \\
&\quad \textit{(ii)} \quad \mathcal{S}_h = \mathcal{S}_\mu \\
&\quad \textit{(iii)} \quad \forall p \in \mathbb{N}_{nu}.\ \mathcal{O}_h(p) = \mathcal{O}_\mu(p) \cup StR_p
\end{aligned}
$$

Thus we have the concurrent simulation framework $(R_{S^n_{\mathrm{MIPS}}, S^n_{\mathrm{MASM}}}, sinv^{\mathrm{MASM}}_{\mathrm{MIPS}})$ for which we need to prove the remaining Assumptions 1 to 3. However without knowing the code generation function asm for MASM statements, we cannot prove Assumption 1 which demands that ownership-safe simulation steps preserve the shared invariant and that ownership-safety can be transferred from the MASM to the MIPS implementation. While we will treat the safety transfer in a separate section, here we can at least give a proof sketch for the preservation of the shared invariant.

PROOF SKETCH FOR PRESERVATION OF $sinv^{\mathrm{MIPS}}_{\mathrm{MASM}}$: We consider a consistency block computation of S^n_{MASM}, $C_\mu \overset{\tau}{\longmapsto} C'_\mu$ that is being simulated by a MIPS computation $H \overset{\sigma}{\longmapsto} H'$. First of all we assume that the safety of (C_μ, τ) can be transfered to (H, σ) and that the same ownership transfer takes place in both sequences, i.e., $\sigma|_{io}.o = \tau|_{io}.o$. This means that we do not transfer any ownership of the stack or code region because it is invisible in MASM.

By the shared invariant on H and C_μ, the shared and owned addresses agree before the (in both cases identical) ownership transfer, therefore they also agree afterwards, giving us claims (ii) and (iii) of the shared invariant. By memory consistency $consis^{mem}_{\mathrm{MASM}}$ from the MASM assembler consistency relation we deduce claim (i). □

With the individual stack regions being owned by the corresponding units, the proofs of the remaining assumptions are straight-forward.

PROOF OF ASSUMPTION 2: We need to prove the following statement for configurations $H, H' \in \mathbb{C}_{S^n_{\text{MIPS}}}$ and $C_\mu, C'_\mu \in \mathbb{C}_{S^n_{\text{MASM}}}$ and some unit $p \in \mathbb{N}_{nu}$.

$$\begin{aligned} &\mathit{consis}_{\text{MASM}}((C_\mu.u_p, C_\mu.m), \mathit{info}_\mu, (H.u_p, H.m)) \\ &\wedge H \approx_p H' \wedge C_\mu \approx_p C'_\mu \wedge \mathit{sinv}^{\text{MASM}}_{\text{MIPS}}(G(H'), \mathit{info}_\mu, G(C'_\mu)) \\ &\qquad\qquad \Longrightarrow \mathit{consis}_{\text{MASM}}((H'.u_p, H'.m), \mathit{info}_\mu, (C'_\mu.u_p, C'_\mu.m)) \end{aligned}$$

By $H \approx_p H'$ we have that the owned and read-only portions of memory on the MIPS level agree between H and H'.

$$\forall a \in H'.\mathcal{O}_p \cup S^n_{\text{MIPS}}.\mathcal{R}.\ H'.m(a) = H.m(a)$$

By the shared invariant we know that the code and stack region for p is contained in this set of addresses, therefore:

$$\forall a \in \mathit{StR}_p \cup \mathit{CR}.\ H'.m(a) = H.m(a)$$

Examining the definition of $\mathit{consis}_{\text{MASM}}$ we see that all sub-relations but memory consistency depend only on those regions of memory. Thus from the consistency relation holding for H and C_μ we obtain all consistency sub-relations besides memory consistency. However the latter is implied by the shared invariant on H' and C'_μ:

$$\forall a \in S^n_{\text{MASM}}.\mathcal{A}.\ H'.m(a) = C'_\mu.m(a)$$

Therefore we conclude that the MASM consistency relation holds for H' and C'_μ. □

PROOF OF ASSUMPTION 3: We need to prove the following statement for configurations $H, H' \in \mathbb{C}_{S^n_{\text{MIPS}}}$ and $C_\mu, C'_\mu \in \mathbb{C}_{S^n_{\text{MASM}}}$ and some unit $p \in \mathbb{N}_{nu}$.

$$\begin{aligned} \mathit{wf}_{\text{MIPS}}(H.u_p, H.m) \wedge H \approx_p H' &\implies \mathit{wf}_{\text{MIPS}}(H'.u_p, H'.m) \\ \mathit{wf}_{\text{MASM}}(C_\mu.u_p, C_\mu.m) \wedge C_\mu \approx_p C'_\mu &\implies \mathit{wf}_{\text{MASM}}(C_\mu.u_p, C'_\mu.m) \end{aligned}$$

We can omit the shared invariant here because the well-formedness of MASM configurations only depends on the program and the stack. Similarly the well-formedness of MIPS configurations only depends on a unit's status register in the SPR file. Therefore well-formedness follows directly from $C_\mu.u_p = C'_\mu.u_p$ and $H.u_p = H'.u_p$. □

Proving Safety Transfer

To discharge Assumption 1 we need to prove that any MIPS computation that is implementing an ownership-safe MASM computation, can also annotated to be safe. The proof of this kind of safety transfer is split into two parts, as we can handle the safety of ownership transfer and the safety of memory accesses separately.

PROOF OF SAFE OWNERSHIP TRANSFER IN ASSUMPTION 1: For a given safe MASM computation $(C_\mu, \langle \tau, o' \rangle)$ that is simulated by $(H.M, \sigma)$, where σ is a consistency block

of some unit p, we need to find an ownership annotation o so that $(D, \langle \sigma, o \rangle)$ is safe. Moreover we need to preserve the shared invariant on the ownership state.

If σ contains only local steps, i.e., $\sigma|_{io} = \varepsilon$, then we also do not transfer ownership on the MIPS level, setting o accordingly, i.e., we do not acquire or release addresses:

$$\forall x \in o.\ x = (\emptyset, \emptyset, \emptyset)$$

Note that such an annotation fulfills the ownership transfer policy and preserves claims (ii) and (iii) of the shared invariant. In the other case, i.e., $\sigma|_{io} \neq \varepsilon$, we know by $one\mathcal{IO}(\sigma, \tau)$ that both sequences contain exactly one $\mathcal{IO}$ step. We denote their positions by the following indices.

$$\exists i \in \mathbb{N}_{|\tau|}.\ \tau_i.io \qquad\qquad \exists j \in \mathbb{N}_{|\sigma|}.\ \sigma_j.io$$

Consequently, we set $o_j = \tau_i.o$ and for all $k \neq j$ we have $o_k = (\emptyset, \emptyset, \emptyset)$. Thus we simply copy the ownership annotation to the corresponing $\mathcal{IO}$ step on the MIPS level. We do not introduce addition ownership transfer on the code or the stack region and the original ownership transfer annotations cannot affect the addresses lying outside of $S^n_{\mathrm{MASM}}.\mathcal{A}$ by construction.

Thus also in this case the shared invariant claims (ii) and (iii) are maintained trivially. Moreover if the ownership transfer was safe on the MASM level, it is also safe for the extended but consistent ownership state of the MIPS machine. □

Note that in the proof above we have not used the fact that on the MASM level all consistency blocks contain only one step as all MASM configurations are consistency points. However this way we can reuse the proof also for the C-IL instantiation. For proving that the MIPS implementation obeys the memory access policy, we would need to know the code generation function of the MASM assembler. However since the shared addresses agree on both simulation layers and we assume that only memory instructions can perform $\mathcal{IO}$ operations, the safety transfer property boils down to a few basic verification conditions on the macro assembler and further arguments in the safety transfer proof.

- All code is placed in the code region and and we do not have self-modifying code, therefore only instructions generated by the macro assembler are executed.
- Plain assembly instructions are translated one-to-one, and we have consistency before and afterwards. Therefore the same addresses are accessed by memory instructions, wrt. to the same ownership state, and safety of memory accesses is preserved.
- This holds especially for $\mathcal{IO}$ steps, because only assembly instructions may perform $\mathcal{IO}$ operations by definition. The implementation of macros does not contain any $\mathcal{IO}$ steps by the $one\mathcal{IO}$ condition on simulating consistency blocks.
- Memory instructions in the translations of macros may only access the stack region, which is always owned by the executing unit, thus they are also safe.

Thus we conclude the MASM section. We have presented semantics for a macro assembly language and introduced its assembler consistency relation. Furthermore we fit it into our frameworks for concurrent systems and simulation. Using the MASM assembler consistency relation we can thus transfer ownership-safety and other verified properties down to the ISA level.

5.6.2 Concurrent C-IL Compiler Consistency

As in the MASM instantiation of our concurrent simulation framework we will first consider the sequential case.

Sequential Simulation Framework

We define the sequential simulation framework $R_{S^n_{\text{MIPS}},S^n_{\text{C-IL}}}$ for $h = (cpu, m) \in \mathbb{L}_{S^n_{\text{MIPS}}}$ and $l \in (s, \mathcal{M}) \in \mathbb{L}_{S^n_{\text{MASM}}}$ as follows.

$$R_{S^n_{\text{MIPS}},S^n_{\text{C-IL}}}.\begin{cases} \mathcal{P} & = & InfoT_{\text{C-IL}} \\ sim(h, info_{IL}, c) & = & consis_{\text{C-IL}}((c.\mathcal{M}, c.s), \pi, \theta, info_\mu, h) \\ \mathcal{CP}a(s, info_{IL}) & = & info_{IL}.cp(s[|s|].f, s[|s|].loc) \\ \mathcal{CP}c(cpu, info_{IL}) & = & (cpu.pc \in A^{\text{C-IL}}_{cp}) \\ wfa(c) & = & wf_{\text{C-IL}}((c.\mathcal{M}, c.s), \pi, \theta) \\ sc(M, t, info_{IL}) & = & sc_{\text{C-IL}}((M.m, M.u(t.s)), t.in, \pi, \theta, info_\mu) \\ wfc(h) & = & wf^{\text{C-IL}}_{\text{MIPS}}(h) \\ suit(\alpha) & = & suit^{\text{C-IL}}_{\text{MIPS}}(\alpha.in) \\ wb(M, t, info_{IL}) & = & wb^{\text{C-IL}}_{\text{MIPS}}((M.m, M.u(t.s)), t.in) \end{cases}$$

Here $A^{\text{C-IL}}_{cp}$ represents the instruction addresses of all consistency-points on the MIPS level and was defined as follows.

$$A^{\text{C-IL}}_{cp} \equiv \{adr(info_{IL}, f, loc) \mid f \in \text{dom}(\mathcal{F}^\theta_\pi) \wedge loc \leq |\mathcal{F}^\theta_\pi.P| \wedge info_{IL}.cp(f, loc)\}$$

Note that this setting of the consistency points coincides with the setting of interleaving-points on both levels. Thus it implies $\mathcal{IPCP}(R_{S^n_{\text{MIPS}},S^n_{\text{MASM}}}, info_\mu)$. For the definition of $\mathcal{IO}$ steps on the MIPS level we again have to define the set A_{io}. Basically we need to collect all addresses of memory instructions which implement volatile variable updates. However without the code generation function we do not know where the implementing memory instruction is placed in the code memory region. To this end we introduce the uninstantiated function $volma$ which returns the instruction address for a volatile memory access at a given location loc of a C-IL function f in program π.

$$volma : Prog_{\text{C-IL}} \times params_{\text{C-IL}} \times InfoT_{\text{C-IL}} \times \mathbb{F}_{name} \times \mathbb{N} \rightharpoonup \mathbb{B}^{32}$$

To compute the function we naturally also need to know the code base address from the compiler information and information on compiler instrinsics from environment

paramter θ. We assume that $volma$ is defined for program locations where we expect volatile variable accesses and for external functions in case they are supposed to update the shared memory.[3] Then A_{io} is defined as follows.

$$A_{io} \stackrel{def}{\equiv} \{volma(\pi, \theta, info_\mu, f, loc) \mid f \in \mathrm{dom}(\pi.\mathcal{F}^\theta_\pi), loc \leq |\pi.\mathcal{F}^\theta_\pi(f).P|\} \setminus \{\bot\}$$

Similar to the MASM instantiation the formulation of the sequential simulation theorem for C-IL in its generalized form is useful for proving the concurrent simulation theorem but not well-suited for getting a grasp of the hypotheses and claims of the MIPS-C-IL simulation. Therefore we reformulate it as follows.

Theorem 10 (Sequential MIPS–C-IL Simulation Theorem) *Given a C-IL starting configuration $c_{IL0} \in \mathbb{C}_{\text{C-IL}}$ that is (i) well-formed, a MIPS configuration $h_0 \in \mathbb{H}_{\text{MIPS}}$ that has external device interrupts disabled and is (ii) consistent to c_{IL0} wrt. some program π, environment parameters θ and compiler information $info_\mu$, while both configurations are in consistency points. Moreover we have an external event vector sequence $eev \in (\mathbb{B}^{256})^*$, that (iii) contains no enabled reset signals. Moreover we demand (iv) that every computation running out of c_{IL0} leads into a well-formed state, that does not produce runtime-errors or stack overflows, and from where the next step does not access the stack or code memory regions. Also (v) the code region fits into memory in the is disjoint from the stack region.*

$$\begin{aligned}
\forall c_{IL0}, \pi, \theta, info_\mu, h_0, eev\ .\quad & \textit{(i)} \quad wf_{\text{C-IL}}(c_{IL0}, \pi, \theta) \wedge h_0.c.spr[\mathrm{dev}] = 0 \\
& \textit{(ii)} \quad consis_{\text{C-IL}}(c_{IL0}, \pi, \theta, info_\mu, h_0) \\
& \qquad \wedge h_0.c.pc \in A^{\text{C-IL}}_{cp} \wedge cp(c_{IL0}, info_{IL}) \\
& \textit{(iii)} \quad \forall v \in \mathbb{B}^{256}.v \in eev \implies v[0] = 0 \\
& \textit{(iv)} \quad \forall c'_{IL} \in \mathbb{C}_{\text{C-IL}}, in \in \Sigma^*_{\text{C-IL}}.\ c_{IL0} \longrightarrow^*_{\delta^{\pi,\theta}_{\text{C-IL}}, in} c'_{IL} \implies \\
& \qquad\qquad wf_{\text{C-IL}}(c'_{IL}, \pi, \theta) \wedge \forall in'.\delta(c'_{IL}, in') \neq \bot \\
& \qquad\qquad \wedge /stackovf(c'_\mu, \pi, \theta, info_{IL}) \wedge msp_{IL} \geq 0 \\
& \qquad\qquad \wedge A^{\pi,\theta}_{c'_{IL}}(stmt_{next}(\pi, c'_{IL})) \cap (CR \cup StR) = \emptyset \\
& \textit{(v)} \quad CR \subseteq [0 : 2^{32}) \wedge CR \cap StR = \emptyset
\end{aligned}$$

If these hypotheses hold then there exists an ISA computation consisting of n steps, that (i) starts in h_0 and is computing according to external event vector sequence eev extended with another external event vector eev'. The computation leads into a state where device interrupts. Similarly (ii) there exists a C-IL computation of m steps running out of c_{IL0} according to some input sequence in. The computation leads into a well-formed C-IL state. Both resulting configurations are (iii) consistency points and the C-IL compiler consistency relation holds between them. Apart from that (iv) the implementing ISA computation does not produce any interrupts nor is it fetching from addresses outside the code region. The implementing computation contains (v)

[3] In case of external function cas, $volma$ returns for location $loc = 1$ the address of the cas instruction implementing the shared memory access. Note that there must exist only one such instruction.

$\mathcal{IO}$ instructions iff the implemented steps of the C-IL machine contains an $\mathcal{IO}$ step and there is (vi) at most one such operation. Let $\mathcal{IO}(c_{IL}, in) = S^n_{\text{C-IL}}.\mathcal{IO}(c_{IL}.s, c_{IL}.\mathcal{M}|_{S^n_{\text{C-IL}}.\mathcal{A}}, in)$ *in:*

$$
\begin{array}{lll}
\Longrightarrow & \exists n \in \mathbb{N}_0, h \in \mathbb{H}^{n+1}, eev' \in (\mathbb{B}^{256})^{n-|eev|}, m \in \mathbb{N}_0, c_{IL} \in \mathbb{C}^{n+1}_{\text{MASM}}, in \in (\Sigma_{\text{C-IL}})^m. & \\
& \textit{(i)} & h_1 = h_0 \wedge h_1 \longrightarrow^n_{\delta_{\text{MIPS}}, eev \circ eev'} h_{n+1} \wedge h_{n+1}.c.spr[dev] = 0 \\
& \textit{(ii)} & c_{IL1} = c_{IL0} \wedge c_{IL1} \longrightarrow^m_{\delta^{\pi,\theta}_{\text{C-IL}}, in} c_{ILm+1} \wedge wf_{\text{MASM}}(c_{\mu_{m+1}}) \\
& \textit{(iii)} & consis_{\text{C-IL}}(c_{ILm+1}, \pi, \theta, info_{IL}, h_{n+1}) \\
& & \wedge\ h_{n+1}.c.pc \in A^{\text{C-IL}}_{cp} \wedge cp(c_{ILm+1}, info_{IL}) \\
& \textit{(iv)} & \forall i \in \mathbb{N}_n.\ /jisr(h_i.c, I(h_i), eev_i) \wedge [\langle h_i.c.pc \rangle : \langle h_i.c.pc \rangle + 3] \subseteq CR \\
& \textit{(v)} & (\exists i \in \mathbb{N}_n.\ h_i.c.pc \in A_{io}) \iff (\exists k \in \mathbb{N}_m.\ \mathcal{IO}(c_{ILk}, in_k)) \\
& \textit{(vi)} & \forall i, j \in \mathbb{N}_n.\ h_i.c.pc \in A_{io} \wedge h_j.c.pc \in A_{io} \implies i = j
\end{array}
$$

Deriving the generalized sequential simulation theorem from this formulization works similar to the MASM case.

Shared Invariant and Concurrent Simulation Assumptions

The shared invariant on ownership and shared memory for the concurrent MIPS–C-IL simulation is similar to that of the MIPS-MASM simulation. There is only a difference concerning the read-only addresses, as these are not empty in C-IL. Nevertheless they are also extended by the code region CR on the MIPS level. Thus we have the following shared invariant.

Definition 87 (Shared Invariant for Concurrent MIPS–C-IL Simulation) *Given memories m_h, m_{IL}, read-only sets $\mathcal{R}_h$, $\mathcal{R}_{IL}$, sets of shared addresses $\mathcal{S}_h$ and $\mathcal{S}_{IL}$, as well as ownership mappings $\mathcal{O}_h$ and $\mathcal{O}_{IL}$, we define the shared invariant for concurrent simulation of $S^n_{\text{C-IL}}$ by S^n_{MIPS} wrt. assembler information $info_{IL}$ as follows. We demand (i) that memory contents are equal for all but the stack and code regions, that (ii) the shared addresses are equal, that (iii) the read-only addresses on the MIPS level contain all read-only addresses from C-IL plus the code region, and (iv) that all units own the same addresses on the MIPS level as on the MASM level plus the individual stack region.*

$$
\begin{array}{l}
sinv^{\text{C-IL}}_{\text{MIPS}}((m_h, \mathcal{S}_h, \mathcal{R}_h, \mathcal{O}_h), info_{IL}, (m_{IL}, \mathcal{S}_{IL}, \mathcal{R}_{IL}, \mathcal{O}_{IL})) \equiv \\
\quad \textit{(i)} \quad m_h|_{S^n_{\text{C-IL}}.\mathcal{A}} = m_{IL} \\
\quad \textit{(ii)} \quad \mathcal{S}_h = \mathcal{S}_{IL} \\
\quad \textit{(iii)} \quad \mathcal{R}_h = \mathcal{R}_{IL} \cup CR \\
\quad \textit{(vi)} \quad \forall p \in \mathbb{N}_{nu}.\ \mathcal{O}_h(p) = \mathcal{O}_{IL}(p) \cup StR_p
\end{array}
$$

Thus we obtain concurrent simulation framework $(R_{S^n_{\text{MIPS}}, S^n_{\text{C-IL}}}, sinv^{\text{C-IL}}_{\text{MIPS}})$ for which Assumptions 1-3 are to be proven. Again, as we do not know the C-IL compilation function, the part of Assumption 1 demanding that the compilation preserves safety wrt. the memory access ownership policy cannot be discharged here. Below we show the part stating that simulating computations preserve the shared invariant. The preservation of ownership transfer safety is proven in the following paragraph.

PRESERVATION OF $sinv^{\text{MASM}}_{\text{MIPS}}$: Since claims (i), (ii), and (iv) are identical to the claims of $sinv^{\text{MIPS}}_{\text{MIPS}}$ their preservation is shown exactly like in the MASM case. Claim (iii) is preserved by simulating computations because $S^n_{\text{MIPS}}.\mathcal{R}$ and $S^n_{\text{C-IL}}.\mathcal{R}$ (which are used as the arguments for inputs $\mathcal{R}_h$ and $\mathcal{R}_{IL}$) as well as CR are constant. □

Also Assumptions 2 and 3 are proven just like for the concurrent MIPS-MASM simulation shown above.

Proving Safety Transfer

It remains to prove that for the MIPS simulation of an ownership-safe C-IL computation we can also find a safe ownership annotation. Since ownership state is extended identically going from C-IL to MIPS as in the MIPS-MASM simulation, also the proof of safe ownership transfer in Assumption 1 is the same.

Concerning memory access safety many of the arguments from MIPS-MASM case also apply to the MIPS–C-IL scenario. Then, using the following facts, we can justify that compiled safe code does not break the memory access policy.

- The compiled code is placed in the code region, which is the only target of instruction fetch for well-behaved code. Moreover we do not have self-modifying code, therefore only instructions generated by the C-IL compiler are executed.
- For any implementation of a C-IL statement only the stack and the memory footprints of the involved expressions may be read or written. Note that this does not follow from compiler consistency for intermediate computation steps.
- Local variables are allocated in the stack region which is owned by the executing computation unit. Therefore if local variable accesses are compiled, so that they access only the stack, the operation is ownership-safe.
- Since there is at most one $\mathcal{IO}$ step per consistency block where ownership can change, the ownership state for the $\mathcal{IO}$ step and all previous local steps is the same and by the shared invariant consistent with the ownership state on the C-IL level before the $\mathcal{IO}$ step. Similarly all successor steps of the $\mathcal{IO}$ operation are computed with the same ownership state that is consistent by the shared invariant with the ownership state after the $\mathcal{IO}$ step on the C-IL level.
- As the same shared memory addresses are accessed wrt. the same ownership state, the ownership-safety of memory access can be transferred.

This finishes our instantiation of the concurrent simulation framework between MIPS and C-IL as well as the chapter on concurrent simulation.

6 Conclusion

In the preceding chapters we have laid out a foundational theory for the construction of pervasive semantic stacks in the formal verification of concurrent computer systems. We started out by proving a general order reduction theorem which allows to reason about concurrent systems considering only a selected subsets of computation schedules where blocks of steps by the same process are interleaved with blocks of other processes at certain *interleaving-points*. These points may be chosen freely by the verification engineer, as long it is guaranteed that there is always at least one interleaving-point between steps by the same process that interact with the environment.

The order reduction was proven for concurrent systems that obey a simple ownership discipline governing shared memory accesses. Following O'Hearn's ownership hypothesis [O'H04], memory is divided into disjoint chunks that are owned by the different processes and in principle a process may only access memory that it owns. In addition there is the shared-writable memory that may be accessed concurrently in predefined synchronization points ($\mathcal{IO}$ steps) using synchronization primitives that access memory atomically. In order to enable more sophisticated synchronization mechanisms, it is allowed to share owned data, that can then be read by other processes in $\mathcal{IO}$ steps. Moreover the ownership state is not stable but may be modified by acquiring and releasing addresses in synchronization points (ownership transfer).

We have shown that ownership-safe steps which obey the ownership policy for memory access and ownership transfer can be reordered across other safe steps, if at least one of them is not an $\mathcal{IO}$ step. This allowed us to give an ownership based proof of the folk theorem that between synchronization points the interleaving of local steps is arbitrary [Bro06]. Using this fact it was quite easy to show that each ownership-safe computation can be reordered into the desired interleaving-point schedule. However for the order reduction to be sound we needed to explicitly transfer the ownership-safety of interleaving-point schedules to arbitrarily interleaved computations the system. Doing so enabled us to represent all computations by the set of interleaving-point schedules and, as opposed to existing order reduction theories [CL98], the order reduction theorem only states hypotheses on the order-reduced system. Since the order reduction argument was made for the operational semantics of a generic shared-memory concurrent computer system (the *Cosmos* model), and the interleaving-points may be chosen with a great flexibility, there are many possible applications of the theorem.

First we showed that the common simplification of coarse scheduling, assumed by concurrent verification tools like VCC, can be justified by setting the interleaving-points to occur before every $\mathcal{IO}$ step. Even more importantly, the order reduction theorem enables to establish pervasive stacks of concurrent program semantics where adjacent

levels are connected formally via simulation theorems. Each level is represented by an individual instantiation of the *Cosmos* model and the interleaving-points are chosen to occur at points where the sequential simulation relation holds for a given process on the concrete level wrt. some configuration of the same process on the abstract level (*consistency points*). Applying the order reduction theorem allowed us to assume blocks of steps by the same process starting in consistency points. We proved an overall *Cosmos* model simulation theorem where we applied sequential simulation theorems on these blocks establishing a system-wide simulation between two concurrent *Cosmos* machine computations. In order to do so we defined generalized formalism for sequential simulation theorems and stated verification conditions that enable the concurrent composition of these sequential theorems. Most prominently for every simulation layer one has to prove an ownership-safety transfer theorem, which requires that ownership-safe abstract computations have ownership-safe implementations on the concrete level. This argument is necessary to transfer the ownership-safety down from the top-level semantics to the lowest level of the semantic stack and apply the order reduction. We demonstrated the applicability of our simulation framework by instantiating it with two concurrent simulations, namely one between a concurrent MIPS instruction set architecture and a concurrent macro assembly semantics, as well as one between concurrent MIPS ISA and a concurrent semantics for an intermediate language of C.

Due to its generality and flexibility, we believe the presented *Cosmos* model model order reduction and simulation theory to be a useful tool in the formal specification and verification of concurrent systems. In the following section we discuss the presented development in more detail, highlighting limitations and possible extensions of the approach. We conclude this thesis by a presentation of future application scenarios.

6.1 Discussion

The presented models and theories are formulated to be as general as possible, spanning a wide range of possible instantiations. However there are some restrictions and limitations of the *Cosmos* model. Some of them are just for simplicity of the presentation, others are necessary for the reordering proofs. An obvious limitation of our model is that we can only have one type of computation units in the *Cosmos* machine. However we can easily extend the *Cosmos* machine definition to contain individual parameters $\mathcal{U}$, $\mathcal{E}$, $reads$, δ, $\mathcal{IO}$, and $\mathcal{IP}$ for each computation unit. As this does not affect the ownership model, the reduction proof will conceptually be unchanged. This allows us to apply the order reduction theorem on even more levels of our model stack for pervasive hypervisor verification. See Sect. 6.3 for examples.

For every *Cosmos* machine instantiation that we consider, the constraint $insta_r$ on the definition of the $reads$-function must pe proven. The constraint can be seen as a correctness condition for the $reads$-function, requiring that it collects all memory addresses that may be read in order to determine the reads-set for the next step of a computation unit. As we have seen in the case of the C-IL semantics, proving this correctness condition may become quite tedious. To overcome the issue, a different constraint for the

reads-set was suggested by W. J. Paul. Since the definition of the reads-set only becomes difficult for systems with serial reads, it should be possible to define the reads-set by iterative application of a series of incremental $reads$-functions, where the first function depends only on the unit state and all following functions depend only on the memory content that was read previously. In case of MIPS ISA, we would only need two incremental $reads$-functions that are applied sequentially, i.e., one function for instruction fetch and another optional function for load instructions. In case of MASM we would only have the $reads$-function for loads, depending on the MASM program and the current program location. For C-IL we would have incremental $reads$-functions for expression evaluation and we would need to determine the order of their application on sub-expressions for the current C-IL statement. If the $reads$-function of a *Cosmos* machine instantiation can be defined using this scheme, then we could drop the requirement to prove $insta_r$, as it is implied by any such definition. However, formalizing and proving this argument is still to be done.

Besides that, another peculiarity of the *Cosmos* model is that we have required the read-only set $\mathcal{R}$ to be fixed, restricting the ownership model from [CS09]. This was necessary in order to keep our reordering proof modular. In fact if the read-only addresses could change dynamically, we could not prove Lemma 21, our basic commutativity lemma that allows permuting safe local steps with others. Imagine we wanted to reorder a safe local step α before an $\mathcal{IO}$ step β that writes and simultaneously releases an address a into the read-only memory. If α is reading a after the release by β, the memory access is perfectly safe according to the ownership-based memory access policy. However, we cannot execute α before β, where a is not yet in the read-only set. Not only would the memory access of α become unsafe, the read access would also have a different result, thus exhibiting a memory race. This does not mean that programs with a dynamic read-only memory are always racy. It just means that we need to conduct a different, more complicated proof to handle such programs. We would need to resort to a monolithic simulation proof like the one in [CS09] and prove the safety of the reordered computation $\alpha\beta$ by assuming the safety of *all* $\mathcal{IP}$ schedules, not only of $\beta\alpha$. We could use this fact to show by contradiction that a situation cannot occur where a is read after a release to the read-only addresses without previous synchronization. In fact Lemma 24 was proven in this style.

In multiprocessor systems the read-only set commonly contains the code region, such that all processors may read from the same code pages without need for synchronization. If we demand the code region to be part of the read-only addresses, then the restriction that the read-only addresses are fixed implies that we may not have self-modifying code. Conversely this means that locations in the code that might be modified may not be marked as read-only. If there is self-modifying code that is not concurrently accessed, then it can simply be owned by the modifying processor. Shared self-modifying code seems to us like a rather obscure phenomenon that should occur only in few places. However we can easily model it by treating fetching from these memory locations, as well as updating them, as $\mathcal{IO}$ steps, ignoring the fact that between modifications they *could* be treated as read-only memory. In other words, if there

is shared self-modifying code in a system, the verification engineer better be very clear about where it is lying, and mark accesses to these instructions accordingly.

Besides explicit modification of the program code in memory, there may be self-modifying code in systems with address translation, due to the swapping of shared code pages. This does not seem as such an exotic scenario after all, although it is quite a 'scary' topic. Assuming processors in a multicore system are fetching code from a shared read-only portion of memory via address translation, then it is advisable to make sure that the entire code is allocated in physical memory. Should this not be the case then page faults can occur whenever there is a fetch from a code page that is not present in physical memory. The fetching processor in this case will enter its page fault handler and load the missing page from a swap memory, e.g. a hard disk. Nevertheless before doing so it should inform other processors that it handles the page fault, to avoid races for page fault handling as well as potential concurrent page fault handling for the code page it is swapping out. The communication with the other processors may be established via inter-processor interrupts (IPI[1]), halting the execution of the environment, or via dedicated data structures in shared memory. Thus the code page swapping can be performed in a sequential manner. In case of the IPI solution, where only a single processor is active, a different *Cosmos* model instantiation can be considered where there is only one processor and the code region is writable. Waking up the other processors, reestablishing the previous model, is then similar to the initialization phase where we would switch from a single core to a multicore *Cosmos* model. For the case where the other processors are not halted, we do not have a proper solution yet. However, one promising idea is to use an extended ownership model where virtual and physical addresses are treated separately. Obviously, further research on this topic is necessary.

Another restriction that is due to the reordering proof is the limitation of the $\mathcal{IO}$ and $\mathcal{IP}$ predicates to depend only on memory content $m|_{\mathcal{R}}$. Of course, when reordering a local step α before some $\mathcal{IO}$ step β this should not turn α into an $\mathcal{IO}$ step. It should stay local. However if the $\mathcal{IO}$ predicate can depend on the shared memory and if α is unsafe, exactly the phenomenon described above can occur. Imagine a MIPS *Cosmos* machine where we define the $\mathcal{IO}$ predicate to cover all executions of cas instructions and step α to fetch a non-cas instruction from a shared memory location a. Naturally, for a local step this is unsafe. If however the previous step β was overwriting a cas instruction at a with that non-cas instruction, reordering α before β would not only make it fetch a cas instruction and thus make it an $\mathcal{IO}$ instruction. The shared memory access would also become safe, thus breaking the coverage of all computations of the *Cosmos* machine by the safe $\mathcal{IP}$ schedules (Lemma 24). Since read-only addresses can not be overwritten by safe steps β, the scenario cannot occur in our model. Moreover

[1] IPIs represent a direct communication between the processors circumventing shared memory. We can model these message channels between processors as special addresses in shared memory, however there are technicalities to be solved, e.g., the processors may not check the channels in shared memory for incoming signals in every step, else all processor steps would need to be considered $\mathcal{IO}$ steps to be ownership-safe. Thus instantiation of the *Cosmos* model with IPIs belongs to future work.

it makes sense to limit the $\mathcal{IO}$ and $\mathcal{IP}$ predicates in this way, because they belong to the specification state reflecting the verification engineer's view of the system. This view in particular contains the knowledge where in a program $\mathcal{IO}$ operations are to be expected, and this expectation should not change dynymically with the program execution. Instead the $\mathcal{IO}$ and $\mathcal{IP}$ predicates should primarily depend on the program code that is executed on the *Cosmos* machine. However in case of processors the code is not stored in the unit state but in memory, therefore the predicates need to be depending on memory. Restricting this dependency on the read-only memories is perfectly fine as long as there is no self-modifying code. If, however, there exists selfmodifying code leading to the generation of $\mathcal{IO}$ steps, we can nonetheless hard-code fetching from the volatile memory location as an $\mathcal{IO}$ step using the program counter. Anyway, such cases seem rather clinical to us, so we will not further elaborate on them.

However there is another difference between our ownership model and the Cohen-Schirmer model from [CS09] connected with the the $\mathcal{IO}$ steps. In the Cohen-Schirmer model ownership may change not only at $\mathcal{IO}$ steps but at any time in the program by inserting ghost instructions that may acquire and release addresses. On the first sight it may not be clear why such an ownership model excludes memory races. If one was free to add ownership transfers to any place in a given computation it is surely possible to make the computation obey the ownership policy, by simply releasing and acquiring the accessed addresses. However this overlooks the fact that [CS09] uses a different formalization of *safety* than we do. In their verification approach, instead of finding annotations for *computations*, one rather has to add annotations to the program code. In a program step these annotations are evaluated resulting in the generation of ghost instructions that modify the ownership state. As it needs to be proven for all interleavings of (annotated) program steps that the ownership policy is obeyed, also this approach with more flexible ownership transfer guarantees memory safety. In our formalism there is no explicit notion of a program that could be annotated (though instantiations might have). To keep the model small and the reordering proof simple, we decided to settle for a more conservative ownership policy, where ownership may only change at $\mathcal{IO}$ points, and a verification apporach where a computations, rather than a program, needs to be annotated with ownership transfer information. Nevertheless, for instantiations that have the concept of a program being executed, this information might stem from annotations that are added to the program code. Apart from these considerations it was shown for the [CS09] model, that the more flexible notion of ownership is equivalent to a more conservative model where addresses are only released in $\mathcal{IO}$ steps, proving that addresses can be released earlier as long as all interleavings are safe. There is a strong conjecture that a similar property can be shown for address acquisition, i.e., proving that addresses can be acquired later. In the proof for releases the ownership transfer is simply mapped to the next $\mathcal{IO}$ step. Similarly, an acquisition of an address would need to be associated with the previous $\mathcal{IO}$ step. In any case it seems like the more flexible ownership transfer is merely syntactic sugar for the verification process, allowing for no additional behaviour in case of programs that are safe for all interleavings. Nevertheless more investigation is necessary, in order to ascertain

this statement for both verification approaches.

Moreover, our definition of *safety* may still be too strict for a number of instantiations, because it demands that *all* $\mathcal{IP}$ schedule computations of a system are safe. We already weakened the hypothesis with the introduction of the *suitability* predicate, however this only allows to exclude certain input sequences. As soon as we want to exclude certain computations from a simulation, we need a stronger notion of suitability that talks also about parts of the system state. For instance in case of the C-IL semantics we needed to modify the input formalism in order to guarantee that there are no run-time errors due to faulty input sequences, allowing to prove the safety of *all* computations. If we had a more expressive suitability predicate, and prove a stronger order reduction theorem accordingly, we could in this case only demand the safety of all computations that are free of run-time errors. However, we would need to prove the Coverage Lemma for the stronger suitability predicate, showing that every suitable computation can be represented by an $\mathcal{IP}$ schedule computation, preserving suitability. This is in general not possible for every kind of suitability predicate because such properties can be broken by reordering for unsafe steps, i.e., an unsafe suitable step might turn into an unsafe unsuitable step and thus not be covered by the safety of all suitable $\mathcal{IP}$ schedules. Therefore we need to find restrictions on the suitability predicates, saying on which parts of the machine state they may depend. For example, if we consider a scenario where the suitability of local steps only depends on the local configuration of the executing unit and the suitability of $\mathcal{IO}$ steps may additionally depend on the shared-writable memory, then suitability is maintained when steps are being reordered across ownership-safe computations. Nevertheless there are suitability predicates that are less conservative. For instance, in a system with separate interrupt handler and interruptable program threads there is a constraint that the interruptable program should not be running while it is interrupted. Only the handler is supposed to be scheduled in this case. However, a predicate that is restricting the set of suitable computations accordingly would need to talk about the states of two computation units at the same time (the interrupted program and the handler), thus not fitting in the conservative scheme outlined above. Hence there is still work left to be done in this area.

Similarly, in the simulation theory, it would be desirable to have a suitability predicate not only on the concrete simulation layer but also on the abstract level. This would facilitate easier instantiations as explained above for the example of C-IL. Furthermore it would allow for the construction of concurrent semantic stacks where the predicate of good behaviour on the concrete layer of an upper simulation level implies the suitability of computations on the abstract layer of the adjacent simulation level below.

Finally, in this thesis we have only described the transfer of safety properties. For the transfer of liveness properties a similar development is needed, using the results of Cohen and Lamport [CL98]. In their work a main difficulty for transferring liveness properties seems to lie in the fact that they are merging blocks of steps into single atomic actions. As we do not do so, we expect that the actual property transfer will become easier. However, we did not prove the order reduction for infinite computations, hence further proofs using coinduction are needed.

6.2 Application

The theory presented in this thesis is supposed to be applied in the pervasive formal verification of concurrent computer systems. For instance, it can be used to prove the correctness of multi-core hypervisors as outlined in [CPS13]. Basically, the order reduction and concurrent simulation theorems are useful whenever one wants to transfer verified properties from an abstract to a concrete semantic layer. Below we want to summarize the steps, the verification engineer needs to carry out in order to verify a safety property P on concurrent specification A and transfer it down to refined specification (or implementation) B.

- First one needs to define operational semantics for layers A and B. Also a simulation relation $consis$ between configuration A and B needs to be stated. It should provide a meaningful translation to property P on level B.
- Next one has to instantiate *Cosmos* models S_A and S_B with the semantics of specification layer A, and B respectively. The $\mathcal{IO}$ steps have to cover all desired interaction between processes on level A. The interleaving-points are determined by the consistency points wrt. $consis$ of implementation B. Then the definitions of $\mathcal{IO}$ steps of B and interleaving-points of A should follow naturally. Nevertheless in choosing the $\mathcal{IO}$ steps and interleaving-points one needs to be careful in order to make sure that $\mathcal{IOIP}$ condition can be fulfilled by the program under verification. Moreover one has to prove restriction $insta_r$ for S_A and S_B.
- The generalized sequential simulation theorem has to be instantiated for simulation of A by B using $consis$. After that the theorem has to be proven. In case a compiler is used to translate A-programs into B-programs, one needs to show the correctness of the code generation here. Also here the verification engineer needs to show that between two consistency points there is at most one $\mathcal{IO}$ step.
- In order to apply the concurrent simulation theorem, one needs to find the shared invariant which couples the shared memory and ownership models between A and B. Naturally, it is useful here to have an idea about how ownership on shared data structures should transfer on level A. Then the refinement of the ownership model to level B is usually easy.
- Having found the shared invariant and suitable ownership models for A and B one has to prove Assumptions 1, 2, and 3. Discharging the latter two assumptions is typically an easy exercise. However the difficultly of the proof of Assumptions 1 depends on the complexity of the shared invariant and the ownership model.
- Now one needs to find an automated verification tool that (provably) verifies ownership-safety on the abstract level A. We believe that VCC [CDH$^+$09] is such a tool, though this proposition has not been proven yet (cf. 6.3). If the verification tool assumes a different granularity of interleaving processes than the consistency

points of S_A, one can apply the order reduction theorem to argue that the verified safety properties hold for all computations of S_A (and thus also for complete consistency block computations).

- Finally the verification engineer proves ownership-safety and the desired safety property P on level A using the automated verification tool. If P translates to predicate $Q[P, par]$ on the concrete *Cosmos* machine S_B, then we can apply Theorem 8 to conclude that $Q[P, par]$ holds on all computations of B.

6.3 Future Work

Besides the improvements of the theory described above there are many tasks remaining where our theorems come into play. First, the formalization of the presented theory in a computer-aided verification tool seems highly desirable. Doing so not only strengthens our confidence in the correctness of the overall theory; it also enables others to use the order reduction theorem in a formal verification context. For instance, the Verified Software Toolchain project [App11] aims at a pervasive formal verification verification methodology building on LeRoy's verified compiler CompCert [Ler09, LABS12]. Since apparently their sequential compiler correctness theorem has not been applied yet in the concurrent case, it would be worthwile to examine applying our concurrent simulation theory in their framework.

Moreover the core part of soundness proofs of separation logics is usually the justification of the parallel decomposition of a concurrent computation into several local computations, where actions of other processes are represented by non-deterministic transitions on the shared resources that occur only in synchronization points. Using our order reduction theorem the soundness proof of this parallel decomposition can be greatly simplified by considering only schedules where other processes are interleaved in synchronization points. Deriving the local computations from the concurrent one for every process is then an easy task. One simply needs to merge subsequent blocks of other steps into a single transition. The same technique may be used to justify the verification approach of tools like VCC. Below we give several further examples for application scenarios of the order reduction and simulation theory.

6.3.1 Device Step Reordering

For a multi-core architecture with devices accessed by memory-mapped I/O, we plan to apply the *Cosmos* model order reduction theorem to reorder device steps to only occur at compiler consistency points of the compiled C+MASM code. This allows us to argue about correctness of device drivers by lifting device models to the C+MASM level, obtaining a semantics of C+MASM with devices. This technique has already been applied in the Verisoft project's device driver verification effort [HIP05], however, the reordering was only shown for single-core processors. For extending it to the multi-core case the *Cosmos* model order reduction theorem seems to be very suitable.

6.3.2 Store Buffer Reduction

Processor models tend to contain store buffers which are first-in-first-out (FIFO) queues that buffer memory write requests. Subsequent reads are served from the local store buffer, if possible – otherwise, memory is accessed.

The ownership model used in the *Cosmos* model model is largely the same as used in the store-buffer reduction theorem of [CS09]. The proof for this theorem is quite complex and contains two implicit reordering theorems, where processor and store buffer steps are reordered with each other and with local steps of other cores. We plan to use the *Cosmos* model order reduction theorem to achieve a simpler proof by using the simplified *Cosmos* ownership model and decomposes the intricate monolithic simulation proof from [CS09] into several smaller, more obvious arguments.

6.3.3 Interrupt Handler Reordering

Interrupts can occur at any point of machine code execution. In order to maintain the concurrent compiler simulation described above, it is useful to reorder interrupts to occur only at compiler consistency points. One first would have to prove a simulation between a *Cosmos* model of interruptible multi-core ISA units and a *Cosmos* model of non-interruptible multi-core ISA and dedicated interrupt handler units. Via order reduction one justifies a model where interrupt handlers are only started in compiler consistency points, allowing the application of the compiler correctness theorem. For verification purposes, interrupt handlers can then be treated just like concurrent C+MASM threads of the system. However there is the aforementioned restriction that the interrupted thread may not be scheduled while the interrupt handler thread is active. Thus in this case one needs a strengthened order reduction theorem that provides safety transfer for suitable subsets of computations.

Moreover there are some technical problems arising when the interrupt handler is using the same stack as the interrupted program. If both entities are modeled as separate computation units, the stack pointer has to be communicated to the handler, which implies that every modification of the stack pointer by the interruptable program must be considered an $\mathcal{IO}$ step. This however leads to problems fulfilling the $\mathcal{IOIP}$ condition of the underlying order reduction. One can avoid the problem by abstracting from the exact value of the stack pointer, using that the interrupt handler uses only a fixed amount of space on the stack and the specification of the handler should not depend on the actual value of the stack pointer it receives. However this argument would need another simulation proof.

Alternatively, one could use the order reduction theorem first and prove a simpler local reordering theorem for the interrupt signals to occur only in consistency points, using software conditions on the handler and the interrupted program. After the application of compiler consistency on these computations splitting the interrupt handler and interrupted program into separate C threads for verification should be easy.

Bibliography

[AHL+09] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging semantics stack for systems verification. *Journal of Automated Reasoning: Special Issue on Operating Systems Verification*, 42, Numbers 2-4:389–454, 2009.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[ALM84] Mack W. Alford, Leslie Lamport, and Geoff P. Mullery. Basic concepts. In Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider, editors, *Advanced Course: Distributed Systems*, volume 190 of *Lecture Notes in Computer Science*, pages 7–43. Springer, 1984.

[App11] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110 – 135, 1975.

[BBBB09] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In Bettina Buth, Gerd Rabe, and Till Seyfarth, editors, *Computer Safety, Reliability, and Security (SAFECOMP 2009)*, volume 5775 of *Lecture Notes in Computer Science*, pages 187–200, Hamburg, Germany, 2009. Springer.

[BG03] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computation Logic*, 4(4):578–651, 2003.

[Bro04] Stephen D. Brookes. A semantics for concurrent separation logic. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK,*

August 31 - September 3, 2004, Proceedings, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004.

[Bro06] Stephen Brookes. A grainless semantics for parallel programs with shared mutable data. *Electron. Notes Theor. Comput. Sci.*, 155:277–307, May 2006.

[CAB+09] E. Cohen, A. Alkassar, V. Boyarinov, M. Dahlweid, U. Degenbaev, M. Hillebrand, B. Langenstein, D. Leinenbach, M. Moskal, S. Obua, W. Paul, H. Pentchev, E. Petrova, T. Santen, N. Schirmer, S. Schmaltz, W. Schulte, A. Shadrin, S. Tobies, A. Tsyban, and S. Tverdyshev. Invariants, modularity, and rights. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics (PSI 2009)*, volume 5947 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 2009.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer. Invited paper.

[CL98] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331, 1998.

[CMST09] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009. Available from `http://research.microsoft.com/pubs`.

[CPS13] Ernie Cohen, Wolfgang Paul, and Sabine Schmaltz. Theory of multi core hypervisor verification. In Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, and Harald Sack, editors, *SOFSEM 2013: Theory and Practice of Computer Science*, volume 7741 of *Lecture Notes in Computer Science*, pages 1–27. Springer Berlin Heidelberg, 2013.

[CS09] Ernie Cohen and Norbert Schirmer. A better reduction theorem for store buffers. Technical report, 2009. arXiv:0909.4637v1.

[Doe77] Thomas W. Doeppner Jr. Parallel program correctness through refinement. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 155–169, 1977.

[DPS09] Ulan Degenbaev, Wolfgang J. Paul, and Norbert Schirmer. Pervasive theory of memory. In Susanne Albers, Helmut Alt, and Stefan Näher, editors,

Efficient Algorithms – Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday, volume 5760 of *Lecture Notes in Computer Science*, pages 74–98. Springer, Saarbrücken, 2009.

[FFQ05] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4):275–291, 2005.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121, 2005.

[Flo67] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[FQ04] Stephen N. Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[GHLP05] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. Melham, editors, *Theorem Proving in High Order Logics (TPHOLs) 2005*, LNCS, Oxford, U.K., 2005. Springer.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GRS05] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of asml. *Theoretical Computer Science*, 343(3):370–412, 2005.

[Gur00] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computation Logic*, 1(1):77–111, 2000.

[Gur04] Yuri Gurevich. Abstract State Machines: An overview of the project. In *Foundations of Information and Knowledge Systems, Third International Symposium, FoIKS 2004, Wilhelminenberg Castle, Austria, February 17-20, 2004, Proceedings*, pages 6–13, 2004.

[HIP05] Mark A. Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pages 309–316, Washington, DC, USA, 2005. IEEE Computer Society.

[HL09] M. Hillebrand and D. Leinenbach. Formal verification of a reader-writer lock implementation in C. In *4th International Workshop on Systems Software Verification (SSV09)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 123–141. Elsevier Science B. V., 2009.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Kle09] Gerwin Klein. Operating system verification — An overview. *Sādhanā*, 34(1):27–69, February 2009.

[KMP14] M. Kovalev, S.M. Müller, and W.J. Paul. *A Pipelined Multi Core MIPS Machine – Hardware Implementation and Correctness Proof*. to appear, draft version on `http://www-wjp.cs.uni-saarland.de`, 2014.

[Kov13] Mikhail Kovalev. *TLB Virtualization in the Context of Hypervisor Verification*. PhD thesis, Saarland University, Saarbrücken, 2013.

[LABS12] Xavier Leroy, Andrew Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Rapport de recherche RR-7987, INRIA, June 2012.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[Lam93] Leslie Lamport. Verification and specifications of concurrent programs. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 347–374, 1993.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LPP05] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*, Koblenz, Germany, 2005.

[LS89] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Technical report, SRC Research Report 44, 1989.

[LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809, Eindhoven, the Netherlands, 2009. Springer. Invited paper.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada*, pages 137–151, 1987.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MP00] S.M. Müller and W.J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[O'H04] Peter W O'Hearn. Resources, concurrency and local reasoning. In *CONCUR 2004-Concurrency Theory*, pages 49–67. Springer, 2004.

[Rus92] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, dec 1992. `http://www.csl.sri.com/papers/csl-92-2/`.

[Sch13a] Sabine Schmaltz. MIPS-86 – A multi-core MIPS ISA specification. Technical report, Saarland University, Saarbrücken, 2013.

[Sch13b] Sabine Schmaltz. *Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C*. PhD thesis, Saarland University, Saarbrücken, 2013.

[Sha12] Andrey Shadrin. *Mixed Low- and High Level Programming Language Semantics and Automated Verification of a Small Hypervisor*. PhD thesis, Saarland University, Saarbrücken, 2012.

[SS12] Sabine Schmaltz and Andrey Shadrin. Integrated semantics of intermediate-language C and macro-assembler for pervasive formal verification of operating systems and hypervisors from VerisoftXT. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *4th International Conference on Verified Software: Theories, Tools, and Experiments, VSTTE'12*, volume 7152 of *Lecture Notes in Computer Science*, Philadelphia, USA, 2012. Springer Berlin / Heidelberg.

[The06] The Santa Cruz Operation Inc. SYSTEM V APPLICATION BINARY INTERFACE – MIPS RISC Processor Supplement 3rd Edition. Technical report, SCO Inc., February 2006.

Index

$\hat{=}$, 36
$\triangleright_p^{blk}$, 157
$[\![\cdot]\!]_c^{\pi,\theta}$, 115
$\lceil\cdot\rceil$, 78
$\lfloor\cdot\rfloor$, 152
$\approx_p$, 42
$\approx_{\overline{p}}$, 42

$\mathcal{A}$, 18
$A_{c_{IL}}^{\pi,\theta}$, 136
A_{code}, 78, 184
A_{cp}, 80, 156, 184, 190
A_{gv}^{θ}, 140
A_{io}, 79, 185, 190, 191
acc_macro, 90
acc_instr, 90
adr, 97, 132
$alloc_{gv}$, 133
array, 107

$bits2bytes$, 118
blk, 152

c, 20
$cas_c^{\pi,\theta}$, 136
c_μ, 85
$codeinv$, 81
$comp$, 25
$conf_{\text{C-IL}}$, 114
$consis_{\text{C-IL}}$, 135
$consis_{\text{C-IL}}^{code}$, 132
$consis_{\text{C-IL}}^{control}$, 132
$consis_{\text{C-IL}}^{lv}$, 134
$consis_{\text{C-IL}}^{mem}$, 133
$consis_{\text{C-IL}}^{regs}$, 132
$consis_{\text{MASM}}$, 99
$consis_{\text{MASM}}^{bp}$, 97
$consis_{\text{MASM}}^{sp}$, 97
$consis_{\text{MASM}}^{code}$, 98
$consis_{\text{MASM}}^{control}$, 97
$consis_{\text{MASM}}^{data}$, 99
$consis_{\text{MASM}}^{mem}$, 99
$consis_{\text{MASM}}^{regs}$, 97
$consis_{\text{MASM}}^{stack}$, 98, 135
cp, 132
$\mathcal{CP}sched$, 161
$\mathcal{CP}sched_c$, 161
$\mathbb{C}_S$, 20

δ, 18
$\delta_{\text{C-IL}}^{\pi,\theta}$, 122
δ_{eret}, 77
δ_{instr}, 73
δ_{jisr}, 76
$drop_{frame}$, 122

$\mathbb{E}$, 110
$\mathcal{E}$, 18

$\mathbb{F}$, 106
$\mathcal{F}_\pi^\theta$, 117
f_{top}, 121
$\mathcal{F}_{adr}$, 106, 109
$\mathbb{F}_{name}$, 106
fp_θ, 136
$frame_{\text{C-IL}}$, 114
$frame_{\text{MASM}}$, 85
$frame_{\text{MASM}}^{no}$, 85
fun, 109
funptr, 107

$FunT$, 112

G_{MIPS}, 64
$\mathbb{G}_S$, 20

$\mathbb{H}_{\text{MIPS}}$, 68

i32, 107
$\mathbb{I}_{\text{C-IL}}$, 111
$\mathbb{I}_{\text{MIPS}}$, 64
inc_{loc}, 119
$info_{\mu}$, 96
$info_{IL}$, 127
$InfoT_{\text{C-IL}}$, 127
$InfoT_{\text{MASM}}$, 96
$insta_r$, 22
$intrinsics$, 117
inv, 27
$\mathcal{IO}$, 18
$\mathcal{IOIP}_{\mathcal{IP}}$, 56
$\mathcal{IO}sched$, 60
$\mathcal{IP}$, 18
$\mathcal{IPCP}$, 161
$\mathcal{IP}sched$, 32
$isarray$, 108
is_empty, 87
$isfunptr$, 108
$isptr$, 108
is_set_up, 86

$\mathbb{K}_{\text{MIPS}}$, 68

$\mathbb{L}_d$, $\mathbb{L}_e$, 155
loc_{top}, 121
lref, 109

m, 20
$\mathcal{M}_{\mathcal{E}\,top}$, 121
$\mathbb{M}_S$, 20
msp_{μ}, 98

nu, 18

$\mathcal{O}$, 20
Ω_S, 24
$oneI\mathcal{O}$, 158

P_{top}, 121
$params_{\text{C-IL}}$, 105
P_g, 180
P_l, 180
$policy_{acc}$, 25
$policy_{trans}$, 26
pop_empty, 89
$prog_{\text{C-IL}}$, 112
ptr, 107

$\mathbb{Q}$, 107
$Q[P, par]$, 182
$\hat{Q}[P, par]$, 181
$qt2t$, 108

$\mathbb{R}$, 155
$\mathcal{R}$, 18
R_{S_d,S_e}, 155
$R_{S^n_{\text{MIPS}},S^n_{\text{C-IL}}}$, 190
$R_{S^n_{\text{MIPS}},S^n_{\text{MASM}}}$, 184
rds_{top}, 121
$reads$, 18
$reqCP$, 131
$R_{\textbf{extern}}$, 106, 117
$\rho|_{io}$, 34
$\rho|_p$, 34

$\mathbb{S}$, 18
$\mathcal{S}$, 20
S_d, 155
S'_d, 175
S_e, 155
S'_e, 165
$safe$, 28
$safe_P$, 55
$safe_{step}$, 28
$safety$, 56
$safety_B$, 153
$safety_{cB}$, 162
$safety_{\mathcal{IP}}$, 56
$safety_{\mathcal{IP}}$, 175
$sc_{\text{C-IL}}$, 137
sc_{MASM}, 100
set_{loc}, 122

set_{rds}, 122
$\Sigma_{\text{C-IL}}$, 120
Σ_S, 24
$\sigma.t, \sigma.o$, 25
sim, 162
$simh$, 175
$sinv$, 162
$size_\theta$, 116
$\mathbb{S}_p$, 18
$Stack_{\text{MASM}}$, 85
$stmt_{next}$, 121
struct, 107
$suit^{\text{MASM}}_{\text{MIPS}}$, 100
$wb^{\text{MASM}}_{\text{MIPS}}$, 100
$swap$, 73

$\mathbb{T}$, 107
$\mathbb{T}_C$, 106
t_{cas}, 136
$\mathbb{T}_Q$, 108
$\tau_{Q_c}^{\pi,\theta}$, 115
θ, 105
Θ_S, 24

$\mathcal{U}$, 18
u, 20
u32, 107
U_c, 161

$\mathbb{V}$, 106
$\mathcal{V}$, 18
val, 108, 109
val, 109
$val2bytes_\theta$, 118
$val_{\textbf{fptr}}$, 109
$val_{\textbf{fun}}$, 109
$val_{\textbf{lref}}$, 109
$val_{\textbf{prim}}$, 108
$val_{\textbf{ptr}}$, 109
vio^{no}_{MASM}, 89
vio^{st}_{MASM}, 90
void, 107
$vol^{\pi,\theta}_{c_{IL}}$, 143
$vol^{\pi,\theta}_{f}$, 130

W, 175
$W^{\pi,\theta}_{c_{IL}}$, 142
$wb^{\text{MASM}}_{\text{MIPS}}$, 100
$wf^{\text{MASM}}_{\text{MIPS}}$, 100
$wf_{\text{C-IL}}$, 116
wf_{MASM}, 87
wfm_{MASM}, 84
$wfprog_{\text{C-IL}}$, 113
$wfprog_{\text{MASM}}$, 84
$write$, 119
$write_{\mathcal{E}}$, 118
$write_{\mathcal{M}}$, 118
$writes_p$, 21

$zero$, 120