Nested Atomic Sections
with Thread Escape:
A Formal Definition

Frédéric Dabrowski
Frédéric Loulergue
Thomas Pinsard

Rapport n° RR-2013-05
Nested Atomic Sections with Thread Escape: 
A Formal Definition

Frédéric Dabrowski Frédéric Loulergue Thomas Pinsard
Univ Orléans, ENSI de Bourges, LIFO EA 4022, Orléans, France

Abstract

We consider a simple imperative language with fork/join parallelism and lexically scoped nested atomic sections from which threads can escape. In this context, our contribution is the precise definition of atomicity, well-synchronisation and the proof that the latter implies the strong form of the former. A formalisation of our results in the Coq proof assistant is also available.

Keywords: atomic sections, well-synchronisation, program traces, formal semantics, proof assistants

1 Introduction

The multi-core trend in architectures development is widening the gap between programming languages and hardware. Improving performances now comes at the price of a deep software renewal because it cannot be done without taking parallelism on board. Unfortunately, current mainstream programming languages fail to provide suitable abstractions to do so. The most common pattern relies on the use of mutexes to ensure mutual exclusion between concurrent accesses to a shared memory. It is widely accepted that this model is error-prone and scales poorly by lack of modularity. In this context, the user is responsible for preserving some sequences of operations from interference. This is typically done by mapping the target (a collection of data) of such operations to a lock to be held when executing the sequence. Different mappings correspond to different choices of granularity, e.g. during a list update one can choose to protect the whole list or simply the updated item and its neighbourhood. A coarse grain helps keeping the code simple, but in general leads to poor performances. On the opposite, a fine grain leads to better performance but the complexity growth is inversely proportional. Despite an important effort of the community to help users in specifying such mappings, mainstream programming languages still do not offer support for doing so mainly because current proposals fail in handling programs in which the mapping changes dynamically. Quoting [20], ownership is in the eye of the Asserter.

Recent research proposes atomic sections as an alternative. In this context, the user simply delimits regions that should be free from interference; the responsibility for ensuring interference freedom is left either to the compiler or to the run-time system. Proposals for implementing atomic sections fall in two categories, depending on the
choice of an optimistic or pessimistic approach to concurrency. The former relies on transactions \[23, 16\], a well established mechanism in database management systems. Intuitively, in this approach, sections are executed optimistically, assuming no interference, but cancelled if any interference occurs. For a discussion on issues raised by the implementation of transactions in a programming language, readers are referred to \[6, 8\]. The latter relies on lock inference \[14, 7\], sections are executed pessimistically, enforcing mutual exclusion between sections by means of locks. With \[5\] we consider that transactions are a mean, an implementation technique, to ensure atomicity. The same remark holds for lock inference and the two approaches could even be combined. For example, lock inference could be used to help a transaction based system to deal with I/O.

At first glance, and independently of the underlying implementation, atomic sections seem more simple to learn compared to more classical synchronisation primitives \[9\]. However, it is not yet clear whether they can be efficiently implemented \[22\] and whether they are really simpler, considering formal semantics and reasoning about programs. Some systems allow the nesting of transactions \[19, 3\] while others do not \[23\], thus leading to poor modularity. When nesting is possible one needs to define precisely the meaning of spawning threads within an atomic section. In \[18\], two primitives for thread creation are proposed: The first one delays the creation until the end of the section (if it is created inside), and the second one forces the thread to live entirely inside the transaction. In \[14\], nesting is allowed but the lock inference scheme prevents an atomic section inside another one to run before the enclosing section terminates when they access the same memory location; because each section must own the lock associated with location. More importantly, one needs to define precisely the meaning of atomicity in this context.

In this paper we do not consider implementation issues but focus on the semantics of atomic sections. We consider a simple imperative language with fork/join parallelism and lexically scoped atomic sections. It supports the nesting of atomic sections and threads are allowed to escape from surrounding sections. Meaning that there is no synchronisation between the end of a section and the termination of threads started within this section. The semantics of the language is as permissive as possible and is not tied to any particular implementation. More precisely, we consider program traces satisfying some basic well-formedness conditions and, more importantly, satisfying the weak atomicity property, i.e there is no interference between concurrent sections. In this context, our contribution is the precise definition of atomicity, well-synchronisation and the proof that the latter implies the strong form of the former (up-to an equivalence relation over traces). A formalisation in Coq \[24, 4\] of our results is available.

We first precise the context of this work with some related work (Section 2). We then describe the semantics domains of our work including the characterisation of well-formed traces (Section 3). We then present our notion of well synchronised traces (Section 4). Section 5 is devoted to our notion of atomicity and to the proof sketch for the main result. We conclude and give future research directions in Section 6. To ease the use of the formalisation of this work in Coq, we give in Section A a correspondance between the definitions and results presented in the paper and their counterparts in Coq.
2 Related Work

Atomic sections have produced many implementations, mainly using transactions, starting from only hardware based [13], to software based [23] [12] [11]. However in general, they did not allow parallelism (and so nesting) in atomic sections. Several papers try to improve these lacks.

Harris [10] proposed to improve composability of atomic sections by adding to them new constructions, like the possibility of specifying another atomic section in case of failure of the first one. But there is still no parallelism or nesting in atomic sections.

The need for nesting was expressed by Moss et al., in [19], by pointing out the example of library use, but also by the fact that atomic sections are less monolithic, and thus lighter. However they restrict their system by disallowing concurrency within an atomic section whereas they still allow it for top level ones. This restriction aims to ease the conflict detection and allows to optimise implementation.

The combination of nested parallelism and nested atomic sections was proposed by Agrawal et al. in [3]. They chose structural parallelism instead of pthread usually used, and thus they rely on a tree representation for the parallelism and atomic sections. Each inner atomic section must be terminated before the atomic section parent completes.

These works focus on efficient implementations. The next ones look over the semantics of atomic sections.

Jagannathan et al. gave operational semantics of a derived version of Featherweight Java with nested and multi-thread atomic sections, independent of any implementation in [15]. They allow nesting but each child atomic section must terminate before its parent. Their work on the proof of atomicity presents some similarities to ours. To prove the correctness of their work, they use program traces to abstract from the operational semantics, and shows the serialisability, i.e. for any abort-free program, there must be a corresponding trace where atomic sections are executed serially.

In [18], Moore et al. pointed out common problems found in several implementations: the precise meaning of atomicity and unnecessary limitations of parallel nesting. To avoid the problem related to the unclear model of atomicity (weak or strong), their advice is to give a semantics with proofs on atomicity. They illustrate it through four languages, and one of them allows parallel nesting with two primitives for thread spawning. The first one spawns threads which must live entirely inside the parent section, and so can only create inner-sections, and the other one delays the spawning at the end of the parent section, and so these threads can only create sub-sections that live outside. We acknowledge the same fact about the unnecessary limitations, and we go further by letting threads escaping from atomic sections.

In [1], Abadi et al. wanted also to face the problem associated with weak and strong atomicity, with the development of transactions based on the automatic mutual exclusion model. In their representation, only one atomic section can be executed at a time. They show that existing implementations of transactions can lead to surprising behaviour due to the weak semantics. They present the semantics of their language, and with some restrictions they can give the behaviour of strong atomicity but with a permissive implementation.
3 Semantics Domain

We consider a kernel imperative language with dynamic creation of threads communicating through a shared memory. Synchronisation of concurrent accesses to this memory is ensured by means of atomic sections which can be nested and support inner-parallelism. Threads forked within a section can escape from its scope, i.e. they may continue their executions after the section terminates. We argue that this asynchronous behaviour of threads, with respect to surrounding atomic sections, benefits programs modularity. This choice contrasts with that of [18] where threads must either complete before the section may terminate, or run after termination of the section (depending on the choice of the primitive used for spawning the thread).

We base our study on partial program traces (Section 3.1) and abstract from program syntax by stating some well-formedness properties (Section 3.2). In particular, we assume that weak atomicity, i.e. non interference of concurrent atomic sections, is ensured by the run-time system through some mutual exclusion mechanism. Because we consider inner-parallelism and thread-escape, the definitions of interference and concurrency between atomic sections require specific care. One must define precisely which threads and atomic sections should be considered as part of an atomic section.

3.1 Traces

We assume disjoint countable sets of memory locations, thread names and section names, elements of which are respectively noted $\ell$, $t$ and $p$, possibly with subscript. The set of values, elements of which are noted $v$, possibly with subscripts, contains at least memory locations, integers and thread names.

**Actions, events and traces.** We define the set of actions, elements of which are noted $a$, possibly with subscripts, as follows.

$$a ::= \begin{align*}
\tau \\
\text{alloc} \ell n \mid \text{free} \ell \mid \text{read} \ell n v \mid \text{write} \ell n v \\
\text{fork} t \mid \text{join} t \mid \text{open} p \mid \text{close} p
\end{align*}$$

Intuitively, $\tau$ denotes an internal, non observable, action. An action $\text{alloc} \ell n$ denotes heap allocation of a block of size $n$ at memory location $\ell$ and an action $\text{free} \ell$ removes such a block from the heap. An action $\text{read} \ell n v$ (resp. $\text{write} \ell n v$) denotes a read (resp. write) access from (resp. to) the offset $n$ from location $\ell$ and $v$ is the read (resp. written) value. Actions $\text{fork} t$ and $\text{join} t$ respectively denote creation and join on a thread $t$. Finally, $\text{open} p$ and $\text{close} p$ respectively denote section opening and closing. Note that section names are purely decorative and have no operational contents, this will be formalised in well-formedness conditions in Section 3.2. Their sole purpose is to name occurrences of atomic sections occurring in traces. An event $e$ is a pair of a thread name and an action, a trace $s$ is a sequence of events.

$$\begin{align*}
\text{(events)}\ e & ::= (t,a) \\
\text{(traces)}\ s & ::= \epsilon \mid s \cdot e
\end{align*}$$
Notations We note \( s_1, s_2 \) the concatenation of partial traces (or traces for short) \( s_1 \) and \( s_2 \) where by abuse of notation \( \epsilon \) stands for \( \varepsilon \cdot e \). For \( i \in \mathbb{N} \), we define the partial function \( \pi_i \) by \( \pi_0(es) = e \) and \( \pi_{i+1}(es) = \pi_i(s) \). We respectively note \( \pi^{\text{act}}(s) \) and \( \pi^{\text{id}}(s) \) the first and second projections over the event \( \pi_i(s) \). We note \( e \in s \), if there exists \( i \) such as \( \pi_i(s) = e \), and by extension \( a \in s \) if \( \pi^{\text{act}}_i (s) = a \).

Definitions To define precisely what should be considered as part of an atomic section, we introduce some auxiliary definitions. Given a trace \( s \), the relations \( \text{owner}_s \) and \( \text{father}_s \) respectively relate a section to its owner thread and a thread to its father. The relation \( \text{range}_s \) denotes the range of a section. By convention, we state that a section \( p \) ranges up to the last position of a trace \( s \) if \( p \) is pending in \( s \). For a well-formed trace \( s \), as defined in Section 3.2, the relations \( \text{owner}_s \), \( \text{father}_s \) and \( \text{range}_s \) will define partial functions.

\[
\begin{align*}
\text{owner}_s p \ t & \triangleq (t, \text{open} \ p) \in s \\
\pi^{\text{act}}_s(i) &= \text{open} \ p \quad \pi^{\text{act}}_s(j) = \text{close} \ p \\
\text{range}_s p & (i,j) \\
\text{father}_s t \ t' & \triangleq (t', \text{fork} \ t) \in s \\
\pi^{\text{act}}_s(i) &= \text{open} \ p \quad \text{close} \ p \not\in s \\
\text{range}_s p & (i,|s|-1)
\end{align*}
\]

It is now possible to define precisely which threads and atomic sections should be considered as part of a section. Given a section \( p \) of a trace \( s \), the relation \( \text{tribe}_s p \) is defined as the least set of thread identifiers containing the owner of the section and threads forked as a side effect of executing the section (relation \( \text{tribeChildren}_s \)).

\[
\begin{align*}
\text{range}_s p & (i,j) \; \; \; \; \; \; \; i < k \leq j \\
\text{owner}_s p t' & \pi_k(s) = (t', \text{fork} \ t) \\
\text{tribeChildren}_s p & t \\
\text{owner}_s p t & \\
\text{tribe}_s p & t
\end{align*}
\]

Intuitively, if \( t \) belongs to \( \text{tribe}_s p \) then the thread \( t \) is part of the computation of the atomic section \( p \) and thus should not be considered as an interfering thread. In the same way, we define a relation over section names stating that an atomic section is part of the computation of another. We say that \( p' \) is a subsection of \( p \) if \( p \subseteq_s p' \), as defined below, holds.

\[
\begin{align*}
\text{range}_s p & (i,j) \; \; \; \; \; \; \; i < k \leq j \\
\text{owner}_s p t & \pi_k(s) = (t, \text{open} \ p') \\
\text{tribeChildren}_s p & t \\
\text{owner}_s p' t & \\
\text{tribe}_s p' & t
\end{align*}
\]

Finally, two atomic sections are said to be concurrent if \( p \bowtie_s p' \), as defined in (1), holds.

\[
(1) \quad p \bowtie_s p' \triangleq p \not\subseteq_s p' \land p' \not\subseteq_s p
\]
### 3.2 Well-formed traces

In this section we state formally some well-formedness conditions over program traces. Those conditions can be seen as a specification for possible implementations of our language. They range from common sense conditions to design choices.

To formalise these conditions we use the following definitions: The predicate $\text{see}_s$ which can be seen as an over-approximation of the information flow in $s$, and is defined as the transitive closure of (2); the relation $\prec_s$ on section names defined at (3).

\[
\begin{align*}
&i < j & \pi_s^{\text{tid}}(i) = t & \pi_s^{\text{tid}}(j) = t \\
&\text{see}_s i j \\
&i < j & \pi_s^{\text{act}}(i) = \text{fork } t & \pi_s^{\text{tid}}(j) = t \\
&\text{see}_s i j \\
&i < j & \pi_s^{\text{act}}(i) = \text{write } \ell n v & \pi_s^{\text{act}}(j) = \text{read } \ell n v \\
&\text{see}_s i j
\end{align*}
\]

(2)

\[
\begin{align*}
&i < j & \pi_s^{\text{act}}(i) = \text{close } p & \pi_s^{\text{act}}(j) = \text{open } p' \\
&\exists i, j. i < j \land \phi_s^{\text{act}}(i) = \text{close } p \land \phi_s^{\text{act}}(j) = \text{open } p' \\
&\text{see}_s i j
\end{align*}
\]

(3)

A trace $s$ is well-formed if it satisfies the conditions in Figure 1 which are explained below:

- Condition (wf1), ensures that section and thread names respectively identify dynamic sections and threads. Conditions (wf2) and (wf3) state simple properties of sections names. Each close action matches a previous open action which should be performed by the same thread. Moreover, each close action of a thread matches the last opened, but not yet closed, section opened by the same thread. As far as section names are concerned, those conditions impose no restrictions over the implementation as section names are purely decorative.

- Conditions (wf4) and (wf5) state usual properties of fork/join instructions.

- Condition (wf6) states that termination of a thread cannot be observed by another thread if the former has pending sections. An implementation can choose either to prevent termination of threads having pending sections or to force closing of such sections on termination. Condition (wf7) states that it is not possible for a thread to join another thread without having explicitly received its name. These conditions ensure that external threads will not interfere with an atomic section by observing termination of inner threads. These conditions match the intuition that atomic section should appear as taking zero-time and thus termination of threads within a section should not be observable before the section is closed.

- Condition (wf8) states that concurrent sections do not overlap.

A trivial result is that the subsection relation safely over-approximates overlapping of atomic sections in well-formed traces. Note that, in the absence of thread escape, the two notions would coincide.
Every action open \( p \), close \( p \) and fork \( t \) occurs at most once in \( s \).

(\( \text{wf}_2 \)) \( \forall i, p. \, \pi_{s}^{act}(i) = \text{close} \, p \implies \exists j. \, j < i \land \pi_{s}^{act}(j) = \text{open} \, p \land \pi_{s}^{tid}(i) = \pi_{s}^{tid}(j) \)

(\( \text{wf}_3 \)) \( \forall p, i, j. \, \text{range}_s \, p \, (i, j) \implies \pi_{s}^{act}(j) = \text{close} \, p \implies \forall k, p'. \, i < k < j \implies \pi_{s}^{tid}(i) = \pi_{s}^{tid}(k) \implies \pi_{s}^{act}(k) = \text{open} \, p' \implies \exists j', k < j' < j \land \pi_{s}^{act}(j') = \text{close} \, p' \)

(\( \text{wf}_4 \)) \( \forall i, t. \, \pi_{s}^{act}(i) = \text{fork} \, t \implies \forall j. \, \pi_{s}^{tid}(j) = t \implies i < j \)

(\( \text{wf}_5 \)) \( \forall i, t. \, (\pi_{s}^{tid}(i) = t \lor \pi_{s}^{act}(i) = \text{fork} \, t) \implies \forall j. \, \pi_{s}^{act}(j) = \text{join} \, t \implies i < j \)

(\( \text{wf}_6 \)) \( \forall p, i, j, t. \, \text{range}_s \, p \, (i, j) \implies \text{owner}_s \, p \, t \implies \forall k. \, \pi_{s}^{act}(k) = \text{join} \, t \implies j < k \)

(\( \text{wf}_7 \)) \( \forall t, i, j. \, \pi_{s}^{act}(i) = \text{fork} \, t \implies \pi_{s}^{act}(j) = \text{join} \, t \implies \text{see}_s \, i \, j \)

(\( \text{wf}_8 \)) \( \forall p, p', \, \text{open} \, p \in \, s \implies \text{open} \, p' \in \, s \implies \text{close} \, p' \in \, s \implies p \triangleleft_s \, p' \lor p' \triangleleft_s \, p \)

Figure 1: Well-Formedness Conditions

**Lemma 1** For all well-formed traces \( s \), sections \( p \) ranging from \( i \) to \( j \) in \( s \) and \( k \) such that \( i \leq k \leq j \) and \( \pi_{s}^{act}(k) = \text{open} \, p' \), for some \( p' \), we have \( p \in_s p' \).

**Sketch of Proof** By hypothesis and conditions (\( \text{wf}_1 \)) and (\( \text{wf}_2 \)) we have neither \( p \not\in_s p' \lor p' \not\in_s p \). From (\( \text{wf}_8 \)) it comes \( p \in_s p' \lor p' \in_s p \). Suppose that \( p \) and \( p' \) are opened by distinct threads \( t \) and \( t' \) (otherwise \( i \leq k \leq j \) entails the result by definition of \( \in_s \)) and \( p' \in_s p \). By definition of \( \in_s \) it comes \( \text{tribeChildren}_s \, p \). It is then immediate that \( \pi_{k}^{act}(s) = \text{open} \, p' \) and \( \pi_{i}^{tid}(s) = t \) imply \( k < i \), thus contradicting the hypothesis. \( \square \)

## 4 Well Synchronised Traces

An important aspect is to identify what kind of isolation is offered by atomic sections. Traditionally two kinds of atomicity can be distinguished [17]. In the weak form the atomic sections are protected only against other sections. It means that instructions outside sections can interfere with data accessed in atomic sections. In this case atomic sections only provide a weak form of protection. On the contrary, with strong atomicity, code inside an atomic section is totally protected both from code in other sections and code outside sections.

As stated in section 3, we rely on the run-time system to ensure some weak-atomicity property: With condition (\( \text{wf}_4 \)) concurrent sections do not overlap. Traditionally, we define a notion of well-synchronisation which provides a sufficient condition for ensuring strong atomicity (as defined in section 5). To do so we define the notion of conflict over actions by the relation \( \neq \) given in Figure 2 and state that a trace is well-synchronised
if a synchronisation occurs between any two events involving conflicting actions. Synchronisation between two events is defined by $sw$ which is the least predicate defined by the transitive closure of rules in Figure 2.

Intuitively, we consider a high-level programming language in which well-synchronisation should not be seen as an additional programming constraint. Indeed, in such languages the user is not expected to deal with non sequentially consistent executions and is responsible for writing data-race free programs. Moreover, thread names and memory locations are assumed to be values of opaque data-types. In particular, there must exist some communication (and thus some synchronisation in well-synchronised traces) between the allocation of a location (resp. the fork of a thread) and any access to that location (resp. join on that thread). Concerning threads names this is ensured by condition $(\text{wt}_7)$. We do not impose an equivalent condition for memory locations here because it is useless to our purpose but in practice we will consider programs satisfying such a property.

Now we can define our property of well-synchronisation, by requiring each conflicting action to be in synchronisation.

**Definition 1** A trace $s$ is well-synchronised if for any conflicting actions $a$ and $a'$ occurring respectively at position $i$ and $j$ in $s$ such as $i < j$, we have $sw_s i j$.

The following lemma states that in well-synchronised traces information cannot flow without synchronisation.

**Lemma 2** For all well-synchronized traces $s$ we have $see_s k k' \Rightarrow sw_s k k'$ for all $k$ and $k'$.
Figure 3: Two examples of ill-formed trace

**Sketch of Proof** Immediate by induction on the proof of see _s k k’_. □

**Examples** We expect, in a trace both well-formed and well-synchronised, that a section is free from outside interference. This motivates well-formedness conditions (w_f_6) and (w_f_7). Without these conditions, traces of Figure [3] would be well-formed and well-synchronised traces but they have indeed a problem of interference. The white and black dots respectively denote fork and join actions. The atomic sections are symbolised by rounded boxes. “w” and “r” respectively represent write and read operations on the same given memory location (and we assume this memory location is not used elsewhere inside section _p_). In both cases the red write operation interferes with the read action on this memory location inside section _p_. Without the join action done by thread _t_0, the two conflicting write operations would not be synchronised. Condition (w_f_7) forbids the join in the first case, and condition (w_f_6) forbids it in the second case.

We can now link our notion of section and synchronisation to express a property on tribes. The following proposition states that in well-synchronised traces members of a section cannot synchronise with non-members while the section is active. An immediate corollary is that in well-synchronised traces actions of members of a section cannot conflict with actions of non-members while the section is active.

**Proposition 1** For all well-formed and well-synchronised traces _s_, sections _p_ ranging from _i_ to _j_ in _s_, and _k, k’_ such that _i_ ≤ _k, k’_ ≤ _j_ and _sw_s k k’_ we have _tribe_s p t_ ⇒ _tribe_s p t’_ where _π_ _s_ ^tid ( _k_ ) = _t_ and _π_ _s_ ^tid ( _k’_ ) = _t’_.

**Sketch of Proof** Let _p_ be a section ranging from _i_ to _j_ in _s_ and let _k’_ be such that
Now suppose that \( i \leq k' \leq j \) and \( p^{\text{tid}}(k') = t' \) for some \( t' \). We consider the property:

\[
P(k_0) \equiv \forall k, t'. \quad i \leq k_0 \leq j \land i \leq k \leq j \land \text{sw}_s k k_0 \land p^{\text{tid}}(k) = t \land p^{\text{tid}}(k_0) = t' \land \text{tribes}_s p t \Rightarrow \text{tribes}_s p t'
\]

Now suppose that \( P(k_0) \) holds for all \( k_0 < k' \). We prove that \( P(k') \) holds by induction on the derivation of \( \text{sw}_s k k' \).

- If \( p^{\text{tid}}(k) = p_{s'}(k') \) or \( p^{\text{act}}(k) = \text{fork}(t') \) the result is immediate by definition of \( \text{tribe} \).

- Suppose that \( p^{\text{act}}(k') = \text{join}(t) \). By \([\text{wf1}]\) we have \( \text{tribeChildren}_{s',p,t} \) and then, by definition of \( \text{tribe} \), we can find some \( k_0 \) and \( t_0 \) such that \( i < k_0 < k' \), \( \text{tribe}_{s,p} t_0 \) and \( p^{\text{act}}(k_0) = \text{fork}(t) \). By \([\text{wf1}]\) it comes \( \text{see}_{s,p} k_0 k' \). By definition of \( \text{see} \), it is easy to check that \( p^{\text{act}}(k') = \text{join}(t) \) implies that there exists some \( k_1 < k' \) such that either \( p^{\text{tid}}(k_1) = t' \) or \( p^{\text{act}}(k_1) = \text{fork}(t') \) and either \( k_0 = k_1 \) or \( \text{see}_{s,k_0} k_1 \). If \( k_0 = k_1 \), then the result is immediate. Otherwise, \( P(k_1) \) holds by induction hypothesis and by \([\text{Lemma2}]\) we have \( \text{sw}_s k_0 k_1 \). By definition of \( \text{tribe} \) it comes \( \text{tribe}_{s,p} t' \).

- Suppose that \( p^{\text{act}}(k) = \text{close}(p') \) and \( p^{\text{act}}(k') = \text{open}(p'') \) for some \( p' \) and \( p'' \). The result is immediate by \([\text{Lemma1}]\).

- The induction step is immediate by applying twice the induction hypothesis.

\[ \square \]

5 Atomicity

In this section, we prove that well-formed and well-synchronised traces satisfy the strong atomicity property. More formally, we prove that well-synchronised traces are serialisable, i.e. any such trace is equivalent to a serial trace. A serial trace \([21]\), is traditionally defined as a trace obtained by a program where atomic sections are executed serially i.e. without interleaving. This definition suits well without inner-parallelism. But in our case an interleaving of “allowed” threads is possible during the execution of the section. That is why we have to define our own notion of seriality. To define the notion of serial trace, we must take care that sections support nesting, inner-parallelism and thread-escape.

This is exactly the purpose of the notion of tribe which captures the set of threads that should be allowed to run while a section is active.

**Definition 2** A trace \( s \) is serial if for all sections \( p \) and positions \( i, j \) and \( k \) such that ranges \( p \) \( (i, j) \) and \( i \leq k \leq j \) we have \( \text{tribe}_s p t \) where \( p^{\text{tid}}(k) = t \).

**Definition 3** A trace \( s \) is serialisable if there exists a serial trace \( s' \) that is equivalent to \( s \).

We now need to define formally what it means for two traces to be equivalent. Usually, traces are defined to be equivalent up to re-orderings preserving the order of conflicting actions, for a broader definition of conflicting actions than the one we use. For technical purpose, we consider a stronger definition of equivalence.
**Definition 4** Two traces $s$ and $s'$ are equivalent, noted $s \equiv s'$, if there exists a bijection $\gamma$ between positions of $s$ and $s'$ such that

\begin{align*}
(e_1) & \quad \pi_s(i) = \pi_{s'}(\gamma(i)) \quad \text{for all } i < |s| \\
(e_2) & \quad sw_s i j \leftrightarrow sw_{s'} \gamma(i) \gamma(j) \quad \text{for all } i, j < |s| 
\end{align*}

**Proposition 2** For all well-formed and well-synchronised traces $s$, and for all traces $s'$, if $s \equiv s'$ then $s'$ is well-formed and well-synchronised.

**Sketch of Proof** The proof of well-synchronisation is done by examining the order between positions in the two traces and exploiting the properties of the bijection to show that these positions are still in synchronisation. The proof of well-formedness is done for each of the eight conditions, using properties of the bijection.

**Theorem 1** Every well-formed and well-synchronised trace is serialisable.

**Sketch of Proof** The proof is by structural induction on $s$. The result is obvious for the empty trace. Suppose $s \cdot (t, a)$ is a well-formed and well-synchronised trace. Then $s$ is both well-formed and well-synchronised because these properties are prefix-closed. By induction hypothesis, there exists $s'$ such that $s' \equiv s$ and $s'$ is serial.

Then we prove that $s' \cdot (t, a) \equiv s \cdot (t, a)$. The bijection used (noted $\gamma$) is the same than in $s \equiv s'$ for all positions $< |s|$ and $\gamma(|s|) = |s'|$. The proof of condition (e₁) of equivalence is trivial. The trickiest part of condition (e₂) is when we have $sw_{s \cdot (t, a)} i |s|$ so we have to prove $sw_{s' \cdot (t, a)} \gamma(i) \gamma(|s|)$. We know that $\pi_s(i) = \pi_{s'}(\gamma(i))$ and $\pi_{s \cdot (t, a)}(|s|) = \pi_{s' \cdot (t, a)}(\gamma(|s|)) = (t, a)$. By an induction on $sw_{s \cdot (t, a)} i |s|$ we can conclude.

By Proposition 2, $s' \cdot (t, a)$ is well-formed and well-synchronised.

Now we show that $s' \cdot (t, a)$ is serialisable. We note exclude$_s p t$ when $p$ is pending in $s$ and $\neg$tribe$_s p t$. Assume $p$ is the left-most section such that exclude$_{s', p} t$.

- Suppose that no such section exists. Let the positions $i, j, k$, the thread name $t'$ and the section name $p'$ be such that range$_{s' \cdot (t, a)} p'(i, j)$, $\pi_{s' \cdot (t, a)}^\text{act}(k) = t'$ and $i \leq k \leq j$. We want to prove that tribe$_{s' \cdot (t, a)} p' t'$. If $k < |s'|$ the result is immediate by the seriality of $s'$. We suppose now that $k = |s'|$ (and then $t = t'$). If $\pi_{s' \cdot (t, a)}(k) = \text{open } p'$ then $k = i$ by (wf₁), and then as the owner of the section $p'$, $t'$ is in the tribe of $p'$. If $\pi_{s' \cdot (t, a)}(k) \neq \text{open } p'$ then necessarily $p'$ is pending in $s'$. By assumption we have thus necessarily that tribe$_{s'} p' t'$ holds. As tribe is preserved by trace concatenation, tribe$_{s' \cdot (t, a)} p' t'$: The trace $s' \cdot (t, a)$ is serial.

- If section $p$ exists, let $i$ be such that $\pi_{s'}^\text{act}(i) = \text{open } p$. We will reason using insertion which is useful to represent the shifting of an element in a trace. Given a non-empty trace $s \cdot e$, we note $s' \gamma_{i_0} e$ the trace obtained by inserting $e$ in $s$ at position $i_0$. A bijection $\gamma$ is induced by the insertion:

$$
\gamma(k) = k \text{ if } k < i \\
\gamma(k) = k + 1 \text{ if } i \leq k < |s| \\
\gamma(|s|) = i_0
$$
Note \( \gamma \) keeps the relative order of positions lower than \(|s|\), and its inverse for positions different from \( i_0 \). There are two important properties of insertion we use:

(\( \text{ins}_{\text{eq}} \)) For all well-formed and well-synchronised trace \( s \cdot (t, a) \),
for all section name \( p \) and position \( i \) such that
- \( \text{excludes}_{s, p} t \) and \( \pi_s^{\text{act}}(i) = \text{open} p \)
- \( \forall k. i \leq k < |s| \implies \neg \text{sw}_{s \cdot (t,a)} k \cdot |s| \)
we have \( s \cdot (t, a) \equiv s \cdot i(t, a) \)

(\( \text{ins}_{\text{tribe}} \)) For all well-formed trace \( s \cdot (t_0, a) \) and position \( i \) such that,
- there exists a section name \( p_0 \) such that \( \text{excludes}_{s, p_0} t_0 \)
- \( \forall k. i \leq k < |s| \implies \neg \text{sw}_{s \cdot (t,a)} k \cdot |s| \)
then for all \( p \) and \( t \), \( \text{tribe}_{s \cdot (t_0,a)} p t \implies \text{tribe}_{s \cdot i(t_0,a)} p t \)

Let us consider the trace \( s'' = s' \cdot i(t, a) \). We now prove that \( s'' \) is equivalent to \( s \cdot (t, a) \) and serial. To do so we need to use the two properties of insertion, so first we prove: \( \forall k. i \leq k < |s'| \implies \neg \text{sw}_{s' \cdot (t,a)} k \cdot |s'| \). Suppose \( \text{sw}_{s' \cdot (t,a)} k \cdot |s'| \) for a \( k \) such that \( i \leq k < |s'| \). By seriability of \( s' \) we know that \( \text{tribe}_{s'} p t_1 \), where \( \pi_{s'}^{\text{tid}}(k) = t_1 \), and \( \text{tribe}_{s' \cdot (t,a)} p t_1 \) by preservation of \( \text{tribe} \) by trace concatenation. By Proposition 1, we can conclude that \( \text{tribe}_{s' \cdot (t,a)} p t \). Because \( s' \cdot (t,a) \) is well-formed we know that \( a \neq \text{open} p \) and \( a \neq \text{fork} t \). By definition of \( \text{tribe} \) we obtain that \( \text{tribe}_{s'} p t \) that contradicts assumption \( \neg \text{tribe}_{s'} p t \).

By (\( \text{ins}_{\text{eq}} \)) and transitivity, we conclude \( s'' \equiv s \cdot (t, a) \).

For proving that \( s'' \) is serial, let \( i', j', k \) be positions, \( p' \) a section name, \( t' \) a thread such that \( \text{range}_{s'} p' (i', j') \), \( i' \leq k \leq j' \) and \( \pi_{s'}^{\text{tid}}(k) = t' \). To prove \( \text{tribe}_{s''} p' t' \) we first prove \( \text{tribe}_{s'} p' t' \) by transposing the \( \text{range}_{s'} p' (i', j') \) on \( s' \) and use the seriability of \( s' \) to conclude. To do so prove that the relative positions of \( i', k \) and \( j' \) are preserved by the inverse of the induced bijection \( \gamma \), i.e. consider whether each position is equal to \( i \) or not: First prove that \( i' \neq i \) and conclude for the four remaining cases using various well-formedness conditions. So \( \text{tribe}_{s'} p' t' \) holds.

As \( \text{tribe} \) is preserved by trace concatenation, we have \( \text{tribe}_{s' \cdot (t,a)} p' t' \) and by (\( \text{ins}_{\text{tribe}} \)), we conclude \( \text{tribe}_{s''} p' t' \). Therefore \( s'' \) is serial.

\[ \square \]

6 Conclusion and Future Work

We based our study on an imperative language with fork/join parallelism and lexically scoped atomic section which supports nesting and parallelism. This parallelism does not impose synchronisation, thus threads can escape from surrounding sections. We design for this language a semantics independent of any implementation. Notions of well-synchronisation and serialisability are defined that allow us to prove that all well-formed and well-synchronised traces are serialisable.
We used the interactive theorem prover Coq to express our definitions and lemmas, and to check our proofs. This formalisation takes roughly 15000 lines with 30% of definitions and proposition statements and the rest of proofs.

We have designed operational semantics of our language with atomic sections and we are checking that a program produces well-formed traces and that a well-synchronised program produces serialisable traces.

This is a part of a larger work where we plan to build a compiler for our language with atomic sections toward a language with only locks. We also plan to use static analysis (like in [2]) to improve our compilation.

Acknowledgements

Thomas Pinsard is supported by a PhD grant from the French ministry of higher education and research. This work is partly supported by ANR (France) and JST (Japan) (project PaPDAS ANR-2010-INTB-0205-02 and JST 10102704).

References


A Formalisation in Coq

The formal development in Coq is available at:

https://traclifo.univ-orleans.fr/PaPDAS/wiki/TransactionsInCoq

In the following tables we give the correspondance between the definitions and lemmas in the paper and their counterparts in Coq. For each one we indicate its Coq name and the Coq file. All the files are in the Transactions directory of the archive.

A.1 Lemma, Propositions and Theorem

<table>
<thead>
<tr>
<th>Name</th>
<th>Coq</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lemma 1</td>
<td>wellFormed_prec_tribe</td>
<td>Trace_Theory.v</td>
</tr>
<tr>
<td>Lemma 2</td>
<td>see_synchronizeWith</td>
<td>Synchronisation.v</td>
</tr>
<tr>
<td>Proposition 1</td>
<td>sw_in_tribe</td>
<td>Synchronisation.v</td>
</tr>
<tr>
<td>Proposition 2</td>
<td>compatible_wellSynchronised</td>
<td>EquivalenceTheory.v</td>
</tr>
<tr>
<td></td>
<td>compatible_wellFormed</td>
<td></td>
</tr>
<tr>
<td>ins_eq</td>
<td>insertion_compatible</td>
<td>SyncInsertion.v</td>
</tr>
<tr>
<td>ins_tribe</td>
<td>insertion_tribe</td>
<td>SyncInsertion.v</td>
</tr>
<tr>
<td>Theorem 1</td>
<td>wsync_atomic</td>
<td>AtomicityFinal.v</td>
</tr>
</tbody>
</table>
### A.2 Definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>Coq</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>section name</td>
<td>t (in module <code>Permission</code>)</td>
<td><code>Permission.v</code></td>
</tr>
<tr>
<td>actions</td>
<td>action</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>events</td>
<td>t (in module <code>Event</code>)</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>traces</td>
<td>Tr</td>
<td><code>GenericTrace.v</code></td>
</tr>
<tr>
<td>∈</td>
<td>sec_order</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>¬</td>
<td>#</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>owns</td>
<td>owns</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>father</td>
<td>father</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>⊲</td>
<td>precedes</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>see</td>
<td>see</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>range</td>
<td>range</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>tribeChildren</td>
<td>tribeChildren</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>tribe</td>
<td>tribe</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>excludes</td>
<td>exclude</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>insertion</td>
<td>insertion</td>
<td><code>Insertion.v</code></td>
</tr>
<tr>
<td>Definition 1</td>
<td>wellSynchronized</td>
<td><code>Synchronisation.v</code></td>
</tr>
<tr>
<td>Definition 2</td>
<td>atomic</td>
<td><code>Atomicity.v</code></td>
</tr>
<tr>
<td>Definition 3</td>
<td>atomic and compatible</td>
<td></td>
</tr>
<tr>
<td>Definition 4</td>
<td>compatible</td>
<td><code>Equivalence.v</code></td>
</tr>
<tr>
<td>$wf_1$</td>
<td>wf_occurences</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>$wf_2$</td>
<td>wf_open_close</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>$wf_3$</td>
<td>wf_seq_order</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>$wf_4$</td>
<td>wf_fork</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>$wf_5$</td>
<td>wf_join</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>$wf_6$</td>
<td>wf_join_all_closed</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>$wf_7$</td>
<td>wf_join_see_fork</td>
<td><code>Trace.v</code></td>
</tr>
<tr>
<td>$wf_8$</td>
<td>wf_mutual_exclusion</td>
<td><code>Trace.v</code></td>
</tr>
</tbody>
</table>