Factorising Temporal Specifications

Marieke Huisman[⋆] **Kerry Trentelman**[†]

⋆ INRIA Sophia Antipolis, France
† RSISE, Australian National University
Marieke.Huisman@sophia.inria.fr
Kerry.Trentelman@anu.edu.au

Abstract

This paper proposes a method to factorise the verification of temporal properties for multi-threaded programs over groups of different threads. Essentially, the method boils down to showing that there exists a group of threads that establishes the property of interest, while the remaining threads do not affect it. We fine-tune the method by identifying for each property particular conditions under which the preservation is necessary. As a specification language we use the so-called specification patterns developed as part of the Bandera project at Kansas State University. For each specification pattern we propose a decomposition rule. We have shown the soundness of each rule using the pattern mappings as defined for LTL. The proofs have been formalised using the theorem prover Isabelle.

Keywords: Specification, program verification, Java, multi-threading, temporal logic

1 Introduction

Over the last few years significant progress has been made in the formal verification of software. Different tools and techniques have been developed that allow one to analyse and verify realistic applications formally; see *e.g.* Burdy et al. (2003), Breunesse, Cataño, Huisman & Jacobs (2003) and Robby, Rodríguez, Dwyer & Hatcliff (2004). However, when it comes to verifying multi-threaded applications, most of these techniques fail to scale up because the verification of a multi-threaded application requires the consideration of all possible interleavings of the different threads, all running in parallel.

To make verification of multi-threaded applications feasible, several techniques can be used to lighten the proof burden. First of all, there are abstraction techniques, which reduce the possible state space of a program; see *e.g* Clarke, Grumberg & Long (1994). Second, there are slicing techniques, removing all those instructions that are irrelevant to the property being checked; see *e.g.* Hatcliff et al. (1999). Finally, recent work proposes the use of atomicity checkers that establish whether the outcome of a method can be affected by the possible interleavings with other threads; see Flanagan & Freund (2004) and Hatcliff, Robby & Dwyer (2004). Any atomic method can be verified in isolation without considering the possible interleavings.

We advocate an alternative approach: in order to simplify the verification tasks we factorise the temporal specifications for the whole system into specifications for a subset of the threads. We do this by defining rules of the form

$$\frac{\mathcal{T}_1 \models \phi \qquad C \models \mathcal{T}_2 \text{ preserves } V}{\mathcal{T}_1 \| \mathcal{T}_2 \models \phi}$$

where ϕ is a temporal specification, $\mathcal{T}_1$ and $\mathcal{T}_2$ are sets of threads, V is a set of variables and C an additional condition under which the variables in V should not be changed. Such rules state that if we wish to verify whether a composed system $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies a temporal property ϕ, it is sufficient to show that one can decompose the system into $\mathcal{T}_1$ and $\mathcal{T}_2$, such that $\mathcal{T}_1$ satisfies ϕ, while $\mathcal{T}_2$ does not affect the validity of ϕ. The latter follows from showing that $\mathcal{T}_2$ preserves a set of variables V which depends on the property ϕ and on the threads in $\mathcal{T}_1$. In this paper we will not discuss in detail how this set V can be constructed, we assume that we have an appropriate dependency analysis available; see *e.g.* Hatcliff et al. (1999).

Our work is motivated by the observation that often one wishes to specify and validate properties only for components of the program, because they are independent of the rest of the program. For example, the behaviour of a bounded channel, receiving messages and delivering them whenever possible, will not depend on the surrounding components that create and consume the messages. Our method allows us to verify the channel thread in isolation, without considering interleavings with the consumer and producer threads. Moreover, if new threads are added to the application, one does not have to redo the verifications, one only has to verify that the new threads do not interfere in an unwanted way, *i.e.* one has to show that under some particular conditions that depend on the property, the new thread does not affect the variables related to the property. For example, suppose we are verifying the property *if the bounded channel is full, it will eventually become non-full*. Our factorisation method allows us to reduce this to (1) the verification of this property for the bounded channel thread, and (2) proving that all other threads do not affect the channel *whenever* the channel is full. The possibility of putting conditions on the factorisation distinguishes our method from, for example, slicing. We believe that checking whether a thread does not affect certain variables can be done efficiently using techniques to check frame conditions; see *e.g.* Spoto & Poll (2003) for a sound method to verify frame conditions.

The model that we use to represent multi-threaded applications is inspired by the programming language Java (Gosling, Joy, Steele & Bracha 2000). There is an arbitrary number of threads, all running in parallel, all using the same shared, global memory. Data can be protected by a lock; only one thread at the time can hold such a lock. The set of possible executions of a Java program is the set of all possible interleavings of the sequential threads, only restrained by the requirements on the locks.

The program model and the temporal specification language have been formalised in Isabelle/HOL (Nipkow, Paulson & Wenzel 2002). Moreover, the proof rules presented below have all been proven correct *w.r.t.* our formalisation. We refer the interested reader to

`ftp://ftp-sop.inria.fr/everest/Marieke.Huisman/Factorisation/` for a full set of theory files.

Related Work Our work is directly inspired by a compositional verification method for Unity (Prasetya & Swierstra 2003). However, the program model of Unity composes non-deterministic processes. Each action in a process is considered to be atomic (even when it has multiple side-effects) and can be executed repeatedly. This is in contrast to our Java program model, where each process is sequential and the level of atomicity is prescribed by the Java memory model (Gosling et al. 2000).

Our work has also been influenced by Santone's compositional approach to verification of concurrent systems, specified using the selective μ-calculus (Santone 2002). However, this approach focuses on processes with synchronised communication, while we concentrate on the shared value model.

As explained above, our work differs from existing approaches to abstraction (Clarke et al. 1994), slicing (Hatcliff et al. 1999) and atomicity checks (Flanagan & Freund 2004, Hatcliff et al. 2004), in that it does not consider the whole application as a single unit. These techniques all aim at reducing the verification burden by eliminating unnecessary verification tasks, while our technique aims at decomposing the program into different parts for which different verification tasks exist.

Finally, we mention compositional model checking and verification approaches; see *e.g.* Clarke, Long & McMillan (1989), Laster & Grumberg (1998) and Sprenger, Gurov & Huisman (2004). These differ from our approach in that they assume an arbitary specification for each component and show that these local specifications are sufficient to ensure the global correctness. In contrast, our approach shows under which conditions it is sufficient to verify a global property only on part of the system.

Notice that our approach does not exclude any of the techniques mentioned above; instead we would advocate a combined use of all the different techniques mentioned.

Organisation of the paper The remainder of this paper is organised as follows. The next section introduces the multi-threaded program model and discusses how this relates to Java. Section 3 introduces the temporal logic that we use to specify program properties. Section 4 discusses the proof rules that we use to factorise temporal specifications, while Section 5 discusses the formalisation and verification of the method. Section 6 shows how our method works in practice. Finally, Section 7 draws conclusions and discusses future work, including possible extensions of our factorisation rules.

2 Program model

We represent programs by labelled transition systems (LTSs). Each thread in the program is represented by a single LTS, the program is represented as their composition. We briefly recall some definitions. Below we will discuss how we assume Java programs to be represented as LTSs.

2.1 Labelled Transition Systems

Definition 1 (LTS) *A* Labelled Transition System (LTS) *is a 4-tuple $\mathcal{T} = (S, A, \rightarrow, I)$, where S is a non-empty set of states, A a set of transition labels, $\rightarrow \subseteq S \times A \times S$ the transition relation denoting whether a state can be reached from another state by an action, and $I \subseteq S$ the set of initial states. For convenience, we write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. We say a transition $a \in A$ is* enabled *in state s, denoted* $\mathsf{enabled}\ \mathcal{T}\ s\ a$*, if there is a state t such that $s \xrightarrow{a} t$.*

Since we model Java programs, we assume that we have a single global shared memory and we only define composition of LTSs with the same state space. The transition relation of the composed LTS is defined as the union of the two individual transition relations, while initial states are defined as the intersection of the individual initial states. This ensures that if both threads initially are enabled, they also will be initally enabled after composition. Formally, the composition of two LTSs is defined as follows.

Definition 2 ($\mathcal{T}_1 \| \mathcal{T}_2$) *Given LTSs $\mathcal{T}_1 = (S_1, A_1, \rightarrow_{\mathcal{T}_1}, I_1)$ and $\mathcal{T}_2 = (S_2, A_2, \rightarrow_{\mathcal{T}_2}, I_2)$ such that $S_1 = S_2$, we define their* composition $\mathcal{T}_1 \| \mathcal{T}_2 = (S, A, \rightarrow_{\mathcal{T}_1 \| \mathcal{T}_2}, I)$, *where*

- $S = S_1 = S_2$
- $A = A_1 \cup A_2$
- $\rightarrow_{\mathcal{T}_1 \| \mathcal{T}_2} = \rightarrow_{\mathcal{T}_1} \cup \rightarrow_{\mathcal{T}_2}$
- $I = I_1 \cap I_2$

Notice that composition is commutative and associative.

Execution traces of the LTSs are infinite sequences of states. Each state in the trace can be reached by a transition from the previous state, or, if there are no more transitions enabled, it is the same as the previous state.

Definition 3 (Trace) *Given an LTS $\mathcal{T} = (S, A, \rightarrow, I)$, we say that the infinite sequence $x = x_0\, x_1\, x_2 \ldots$ of states is a* trace *of $\mathcal{T}$, written* $\mathsf{trace}\ \mathcal{T}\ x$*, if*

- $x_0 \in I$ *and*
- *for all i,*
  if there exists an $a \in A$ such that $\mathsf{enabled}\ \mathcal{T}\ x_i\ a$*,*
  then there exists an $a' \in A$ such that $x_i \xrightarrow{a'} x_{i+1}$,
  otherwise $x_i = x_{i+1}$.

If there are no more transitions enabled we say a trace is stuttering*. In particular, we use* $\mathsf{stutters}\ x_i$ *to denote that the trace stutters from x_i onwards,* i.e. $\forall j.\ j \geq i.\ x_j = x_i$.

We say a finite sequence $x_0, \ldots, x_n$ is an initial trace fragment up to n*, denoted* $\mathsf{trace\_upto}\ T\ x\ n$*, if for all i between 0 and n it satisfies the conditions above.*

We use $\mathsf{trace\_pred}\ \mathcal{T}\ x$ to denote that the infinite sequence satisfies the second condition of the definition above. Notice that we have

$$\mathsf{trace}\ \mathcal{T}\ x \Leftrightarrow \mathsf{trace\_pred}\ \mathcal{T}\ x \wedge x_0 \in I_{\mathcal{T}}$$

We adopt the notation of Emerson (1990) such that x^j denotes the suffix trace $x_j, x_{j+1}, x_{j+2}, \ldots$ and x_i^j denotes the segment trace $x_i, x_{i+1}, \ldots, x_{j-1}, x_j$.

Below we sometimes use a generalisation of traces, denoted $\mathsf{trace\_q}\ q\ \mathcal{T}\ x$, where we require the first state to satisfy an arbitrary predicate q (instead of simply requiring that it is an initial state). Hence we write

$$\mathsf{trace\_q}\ q\ \mathcal{T}\ x \Leftrightarrow q(x_0) \wedge \mathsf{trace\_pred}\ \mathcal{T}\ x$$

Notice that this immediately gives us

$$\mathsf{trace}\ \mathcal{T}\ x \Leftrightarrow \mathsf{trace\_q}\ (\in I)\ \mathcal{T}\ x$$

Finally, we will assume that all executions are fair, *i.e.* no transition can be enabled forever without being executed. Fairness is defined as follows: if a transition is enabled, then eventually it will happen, otherwise it will become disabled.

Definition 4 (Fairness) *Given an LTS $\mathcal{T}$ and infinite sequence x such that* $\mathsf{trace}\ \mathcal{T}\ x$*, we say x is* fair *if for all i and a such that* $\mathsf{enabled}\ \mathcal{T}\ x_i\ a$*, there exists a $j \geq i$ such that either* $\neg\mathsf{enabled}\ \mathcal{T}\ x_j\ a$*, or* $x_j \xrightarrow{a} x_{j+1}$.

All definitions have been formalised in Isabelle/HOL and several useful results have been proven. Of these, the most interesting result is that any initial trace fragment can be extended to a full trace fragment by arbitrarily picking enabled transitions until no transitions are enabled anymore. This is exhibited in the following equation which states that for every x that is an initial trace fragment up to j, we can find a trace x' that coincides with x on the first j elements.

$$\begin{array}{l}\mathsf{trace\_pred\_upto}\ \mathcal{T}\ x\ j \Rightarrow \\ \quad \exists x'.\ \mathsf{trace\_pred}\ \mathcal{T}\ x' \wedge (\forall i.\ i < j \rightarrow x_i = x'_i)\end{array} \quad (1)$$

2.2 Modeling Java

As mentioned above, our program model is inspired by the programming language Java. Following the Java language specification (Gosling et al. 2000), an execution of a (multi-threaded) Java program can be seen as a sequence of memory actions. These memory actions can be lock, unlock, write, read, store, load, assign and use, all with appropriate parameters. These actions cannot occur in arbitrary order: a set of rules exist that restrict the possible interactions; see Gosling et al. (2000) and Cenciarelli, Knapp, Reus & Wirsing (1999). Typically, we would assume that these actions are the labels of our transitions (the state space is modelled as a mapping from variables to values). However we never make this explicit, we only assume that identically labelled transitions have identical effects on the state space:

$$\begin{array}{l}s \xrightarrow{a} t \wedge s' \xrightarrow{a} t' \Rightarrow \\ \quad \forall x.\ s(x) = s'(x) \Rightarrow \\ \quad\quad s(x) \neq t(x) \wedge s'(x) \neq t'(x)\end{array} \quad (\text{Ass } 1)$$

This assumption states that if we have two identically labelled transitions $s \xrightarrow{a} t$ and $s' \xrightarrow{a} t'$, then if s and s' coincide on x, then if the first transition will change the value of x (*i.e.* $s(x) \neq t(x)$), then the second transition will also change the value of x, thus $s'(x) \neq t'(x)$.

In fact, in Java, each thread also has a private memory. We could explicitly incorporate this, but this is not strictly necessary: it is sufficient to assume that certain parts of the global memory will only be changed by a single thread. We assume that all threads are already created and can be represented by a single LTS[1].

3 Temporal Formulae

We use the specification patterns as originally proposed within the Bandera project at Kansas State University as a property specification language; see Dwyer, Avrunin & Corbett (1998) and the specification patterns website[2]. Specification patterns describe the most common constructs found in temporal logic specifications. For each pattern a mapping into different logics, such as LTL (Emerson 1990), CTL (Emerson 1990) and regular alternation-free μ-Calculus (Mateescu 1998) is defined. The aim of the specification patterns project is to make it easier and more intuitive to write temporal system specifications.

Basically, each pattern describes a property that has to hold in a certain region of the system execution. This region is called the *scope* of the pattern. Two kinds of properties are distinguished: occurrence patterns (absence, universality and existence) and order patterns (responds-to and precedes). Figure 1 shows the Isabelle datatypes that formalise the patterns (where `'s` is a polymorphic type representing the state).

```
types 's pred = "'s => bool"

datatype 's scope =
   Globally
 | After "'s pred"
 | Before "'s pred"
 | Between "'s pred" "'s pred"
 | AfterUntil "'s pred" "'s pred"

datatype 's pattern =
   Universal "'s pred" "'s scope"
 | Absent "'s pred" "'s scope"
 | Exists "'s pred" "'s scope"
 | RespondsTo "'s pred" "'s pred"
                              "'s scope"
 | Precedes "'s pred" "'s pred"
                              "'s scope"
```

Figure 1: Isabelle formalisation of specification patterns

To give a semantics to these specification patterns, we use the mapping of the patterns into LTL as defined on the specification patterns website – we call this mapping pat2ltl – and formalise the semantics of LTL – following (Emerson 1990) – in Isabelle. We say $\mathcal{T}$ satisfies property ϕ, denoted $\mathcal{T} \models \phi$, if for all infinite sequences x such that $\mathsf{trace}\ T\ x$ we have $x \models_{\mathsf{LTL}} \mathsf{pat2ltl}(\phi)$, where $\models_{\mathsf{LTL}}$ corresponds to the usual satisfaction relation of LTL-formulae. Alternatively, we sometimes write $\mathcal{T}, q \models \phi$ if all traces starting with property q satisfy the formula ϕ, *i.e.* for all infinite sequences x such that $\mathsf{trace\_q}\ q\ T\ x$ we have $x \models_{\mathsf{LTL}} \mathsf{pat2ltl}(\phi)$. We call this *q-satisfaction*.

Notice that to show that a property ψ with a scope $\mathsf{Between}\ q\ r$ holds on $\mathcal{T}$, it is sufficient to show that $\mathcal{T}$ q-satisfies $\psi\ \mathsf{Between}\ q\ r$.

$$\mathcal{T}, q \models \psi\ \mathsf{Between}\ q\ r\ \Rightarrow\ \mathcal{T} \models \psi\ \mathsf{Between}\ q\ r \quad (2)$$

Similar results can be proven for the scopes $\mathsf{After}\ q$ and $\mathsf{AfterUntil}\ q\ r$.

Whilst formalising the semantics, we found some small ambiguities in the mappings (missing brackets, *etc.*) and we encountered one major problem. Following the website, the pattern $p\ \mathsf{Precedes}\ q\ (\mathsf{After}\ r)$ is mapped into the LTL-formula $[\,]\neg r \vee \langle\,\rangle\,(r \wedge (\neg q\ \mathsf{W}\ p))$ (where W is the weak until operator, *i.e.* $p\ \mathsf{W}\ q$ means p holds until q, or p holds forever). This formula says that either r never holds, or there is a place where r holds and from that point on q will not hold, unless p has held before.

However, notice that this mapping does not require that the property $\neg q\ \mathsf{W}\ p$ holds after the first time r is true, it only requires it to hold some time after r is true. This implies that the mapping would accept the following trace:

Here, the property $\neg q\ \mathsf{W}\ p$ holds after the second time r is true. This is because in the next state p holds and q becomes true only one state later [3]. In our opinion this trace should be considered incorrect, because after the first occurrence of r, q occurs without a preceding occurrence of p. Therefore we changed the mapping of $p\ \mathsf{Precedes}\ q\ (\mathsf{After}\ r)$ into

$$\neg r\ \mathsf{W}\ (r \wedge (\neg q\ \mathsf{W}\ p))$$

[1] Our model would allow dynamic thread creation, however, this would make the separation in different threads more involved.

[2] `http://patterns.projects.cis.ksu.edu/`

[3] We assume only the properties mentioned in the states to hold; all other properties are false.

This rejects the trace above and corresponds better to our intuition of the meaning of this pattern. Moreover, this closely resembles the mapping of this pattern into a CTL formula.

4 The Factorisation Rules

As explained above, our aim is to factorise the verification of temporal properties over the threads in a program. Given a program and a temporal property, we divide the different threads in the program into two groups: for the threads in the first group we show that they establish the property, for the threads in the other group we show that they do not affect it. We assume that each thread is modelled by a labelled transition system and that a program consists of a collection of threads. However, since our LTS composition operator is associative and commutative, it is sufficient to provide factorisation rules for the composition of two LTSs $\mathcal{T}_1 \| \mathcal{T}_2$.

Preservation To show that an LTS does not affect a temporal property, we require that it does not change any variable that is related to the property. Typically, these related variables will be all the variables that are mentioned in the property and any variable on which these variables (directly or indirectly) depend. However, in general it is not necessary that the second LTS always preserves the set of variables. For each temporal property we can state the precise conditions under which this set of variables has to be preserved. Moreover, in some cases the second thread might also make a step which does not preserve the set of variables, but actually makes the temporal property hold for the composed system. For example, to show that a property Exists p Globally holds on a composed system – *i.e.* on every path there always exists a p – it is sufficient to show that the second group of threads preserves the set of variables on which p depends, unless it makes p true. To be as general as possible, our factorisation rules allow the second group of threads, wherever possible, to make the property hold for the composed system.

Before formally defining preservation, we first define equality of states *w.r.t.* a set of variables V. As explained above in Section 2, we consider states as mappings from variables to values. We consider that two states are V-*equal* if they coincide on the values of all variables in V.

Definition 5 (V-equality) *Given states s and t and a set of variables V, we define* V-equality *between s and t, denoted $s =_V t$, as follows:*

$$s =_V t \Leftrightarrow (\forall v.\ v \in V \Rightarrow s(v) = t(v))$$

Notice that V-equality is reflexive, symmetric and transitive. Further, it is preserved by the subset-relation. Notice that V-equality *w.r.t.* the empty set reduces to true, while V-equality *w.r.t.* the universal set reduces to standard equality[4].

$$\begin{array}{rcl} & & s =_V s \\ s =_V t & \Leftrightarrow & t =_V s \\ s =_V t \wedge t =_V u & \Rightarrow & s =_V u \\ s =_V t \wedge W \subseteq V & \Rightarrow & s =_W t \\ s =_\emptyset t & \Leftrightarrow & \mathsf{true} \\ s =_\mathcal{V} t & \Leftrightarrow & s = t \end{array}$$

Now we are ready to define preservation.

Definition 6 (Preserves) *Given an LTS $\mathcal{T}$, a set of variables V and state predicates p and q, we define* preservation *as*

$$p \models \mathcal{T} \text{ preserves } V \mid q \Leftrightarrow \\ \forall s\, t\, a.\ p(s) \Rightarrow s \xrightarrow{a} t \Rightarrow (s =_V t \wedge p(t)) \vee q(t)$$

[4] We use $\mathcal{V}$ to denote the universal set of variables.

Thus when p holds in a state s, all states that are directly reachable from this state should either preserve V or make q hold.

Notice that it is straightforward to prove that preservation is preserved by the subset relation.

$$U \subseteq V \Rightarrow \\ p \models \mathcal{T} \text{ preserves } V \mid q \Rightarrow p \models \mathcal{T} \text{ preserves } U \mid q$$

Here we do not go further into how the preservation property can be checked, but we believe that this can be done relatively easily using existing standard techniques for program verification, for example, those used to check assignable clauses of JML specifications (Cataño & Huisman 2003, Spoto & Poll 2003).

Dependency sets As explained above, for each temporal property we define the set of variables that have to be preserved. This set of variables depends on the variables used in the different state properties and on the program, or more accurately, on the program's dependency graph. Again, we do not go into details about dependency analysis, but we refer *e.g.* to Hatcliff, Dwyer & Zheng (2000) for an identification of the different dependencies that can occur in a multi-threaded Java program. We use $\mathsf{dep}_p\ \mathcal{T}$ to denote the set of variables on which the variables in p depend *w.r.t.* the program represented by $\mathcal{T}$. We assume that our dependency analysis distributes over conjunction and negation.

$$\mathsf{dep}_{p \wedge q}\ \mathcal{T} = \mathsf{dep}_p\ \mathcal{T} \cup \mathsf{dep}_q\ \mathcal{T} \qquad \text{(Ass 2)}$$

$$\mathsf{dep}_{\neg p}\ \mathcal{T} = \mathsf{dep}_p\ \mathcal{T} \qquad \text{(Ass 3)}$$

For convenience we write $\mathsf{dep}_{p,q}\ \mathcal{T}$ to denote $\mathsf{dep}_p\ \mathcal{T} \cup \mathsf{dep}_q\ \mathcal{T}$.

Furthermore, we assume that the dependency relation properly coincides with the transition relation: meaning that for any transition $s \xrightarrow{a} t$ that does not preserve the variables in the dependency set, and for any state s' that is equivalent to s *w.r.t.* the dependency set, we can find a state t' that can be reached with a similar transition and is equivalent to t *w.r.t.* the dependency set. This ensures that the dependency set is closed and that there are no "leaking" dependencies. In other words, if we would reduce the state space to V the transitions are indistinguishable. Formally, we specify this assumption as follows:

$$\begin{array}{l} s \xrightarrow{a} t \Rightarrow s \neq_{\mathsf{dep}_p\ \mathcal{T}} t \Rightarrow \\ \quad \forall s'.\ s =_{\mathsf{dep}_p\ \mathcal{T}} s' \Rightarrow \\ \quad\quad \exists t'.\ s' \xrightarrow{a} t' \wedge t =_{\mathsf{dep}_p\ \mathcal{T}} t' \end{array} \qquad \text{(Ass 4)}$$

A last important assumption concerning the dependency set is that if two states s and t are equivalent *w.r.t.* to the set $\mathsf{dep}_p\ \mathcal{T}$, then p holds in s implies p holds in t (and *vice versa*, which follows directly from the symmetry of the V-equality relation).

$$s =_{\mathsf{dep}_p\ \mathcal{T}} t \Rightarrow p(s) \Rightarrow p(t) \qquad \text{(Ass 5)}$$

The rules Finally we are ready to present the different rules. In the remainder of this section we present the intuition for two rules ($\phi = \mathsf{Exists}\ p\ \mathsf{Between}\ q\ r$ and $\phi = p\text{-}\mathsf{RespondsTo}\ q\ \mathsf{After}\ r$). The next section discusses our verification strategy and presents the proof of one of these rules in detail.

We refer to the appendix for an overview of the rules for universality, existence, precendence and response. The rules for absence can be directly derived from the rules for universality by using the following equivalence (where γ denotes an arbitrary scope).

$$\mathcal{T} \models \mathsf{Absent}\ p\ \gamma \Leftrightarrow T \models \mathsf{Universal}\ \neg p\ \gamma$$

Exists p Between $q\,r$ If we would like to show that a composed LTS $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies a property ϕ = Exists p Between $q\,r$, it is sufficient to show that (1) every trace in $\mathcal{T}_1$ that starts with a state satisfying q satisfies the property, *i.e.* $\mathcal{T}_1$ q-satisfies ϕ, and (2) $\mathcal{T}_2$ preserves the properties p and r until p is true.

$$\frac{\mathcal{T}_1,\ q \models \phi \quad \neg p \wedge \neg r \models \mathcal{T}_2 \text{ preserves } (\mathsf{dep}_{p,r}\ \mathcal{T}_1) \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \phi}$$

To understand that this is sufficient, suppose we have a trace x of $\mathcal{T}_1 \| \mathcal{T}_2$ on which there is a state where q is true. If r eventually holds later on this trace, then we have to show that p holds earlier, otherwise the property trivially holds. The first hypothesis of the proof rule states that $\mathcal{T}_1$ ensures this, *i.e.* for any trace of $\mathcal{T}_1$ starting with q, if r eventually holds then p holds before. The second hypothesis ensures that any trace of $\mathcal{T}_1 \| \mathcal{T}_2$ between q and r can actually be considered equivalent to a trace of $\mathcal{T}_1$, because $\mathcal{T}_2$ does not disturb it, *i.e.* as long as p and r are not true, $\mathcal{T}_2$ will not change their validity. There is one exception to this, namely $\mathcal{T}_2$ is allowed to make p true itself. In this case the dependency sets of p and r no longer have to be preserved, because it is sufficient to have a single existence of p.

p RespondsTo q After r Similarly, if we wish to show that a composed system $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies a property ϕ = p RespondsTo q After r, *i.e.* if after r somewhere q holds, then eventually p will hold, it is sufficient to show that (1) any $\mathcal{T}_1$ trace starting with a state satisfying r satisfies ϕ, and (2) as long as $\neg p$ holds, $\mathcal{T}_2$ preserves p and q unless it makes p true.

$$\frac{\mathcal{T}_1,\ r \models \phi \quad \neg p \models \mathcal{T}_2 \text{ preserves } (\mathsf{dep}_{p,q}\ \mathcal{T}_1) \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \phi}$$

The validity of this rule can be intuitively understood as follows. Suppose we have a trace in $\mathcal{T}_1 \| \mathcal{T}_2$ for which we want to show p RespondsTo q After r and suppose r holds on this trace. The first hypothesis tells us that any $\mathcal{T}_1$ trace starting with r satisfies the property. The second hypothesis tells us that any trace of $\mathcal{T}_1 \| \mathcal{T}_2$ can be considered equivalent to a $\mathcal{T}_1$ trace up to the moment p holds, because $\mathcal{T}_2$ does not change any of the relevant variables. As in the example above, as an exception $\mathcal{T}_2$ can make p hold, in which case the formula on the composed system trivially holds.

5 Formalisation and correctness

As mentioned above, all rules have been formalised and proven correct using Isabelle (Nipkow et al. 2002). This section sketches the correctness proof of one such rule – for Exists p Between $q\,r$ – in more detail. Similar proofs have been constructed for all other rules. However, first we will sketch the general approach we used for verification of the rules.

All factorisation rules have the following shape (where C_1, C_2 and C_3 are arbitrary boolean formulae containing the atomic propositions in ϕ).

$$\frac{\mathcal{T}_1 \models \phi \qquad C_1 \models \mathcal{T}_2 \text{ preserves } (\mathsf{dep}_{C_2}\ \mathcal{T}_1) \mid C_3}{\mathcal{T}_1 \| \mathcal{T}_2 \models \phi}$$

Suppose that we would like to show the correctness of such a rule. In order to do this we have to show that for an arbitrary trace of $\mathcal{T}_1 \| \mathcal{T}_2$ the property $\mathsf{pat2ltl}(\phi)$ holds. Intuitively, if we have a trace x of a composed LTS, we should be able to find an equivalent trace of the isolated $\mathcal{T}_1$ system, because $\mathcal{T}_2$ is guaranteed not to affect the variables relevant to the property[5]. However x can contain arbitrary transitions that are irrelevant *w.r.t.* the property ϕ. These transitions can be made both by $\mathcal{T}_1$ and $\mathcal{T}_2$. Therefore we use slicing to construct (an initial fragment of) a trace y from x, and we show that y is (an initial fragment of) a trace of $\mathcal{T}_1$. Since we know that $\mathcal{T}_1$ satisfies ϕ and $\mathcal{T}_2$ preserves the appropriate set of variables, we can conclude that $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies ϕ.

The main challenge in the verification of the different rules is the proof that one can construct the (initial fragment of a) trace of $\mathcal{T}_1$. Essentially, we need to show that if a state s is enabled in the sliced $\mathcal{T}_1$ trace, then a transition is made, otherwise the trace stutters. We relate s to a state s' in the original trace and consider the different possibilities for s' in the original trace:

- s' is stuttering;
- s' makes a transition to a state that is equivalent *w.r.t* the slicing relation, which is thus not visible in the sliced trace; or
- s' makes a transition to a state that is different *w.r.t.* the slicing relation, and which is thus visible in the sliced trace.

In the latter case, we have to distinguish whether the transition is made by $\mathcal{T}_1$ or $\mathcal{T}_2$ (the latter of these usually leads to a contradiction). These proofs are long and involved and we will not go into further details here.

Below, we first introduce appropriate definitions and assumptions that we use for slicing, and then we discuss the proof of the rule Exists p Between $q\,r$ in more detail.

Slicing As explained above, in our proofs we use the notion of slicing to remove irrelevant transitions from our LTSs. In addition, we define a function that given a normal trace, produces a trace of the sliced LTS.

Ordinarily, slicing techniques are applied to code, removing those parts of a program that are irrelevant to the property being verified. The sliced program is then modelled by a smaller, more manageable LTS; see Hatcliff et al. (2000) for an automated approach to slicing Java programs. In our model however, rather than slicing the program, we slice the LTSs themselves.

In our verifications, we assume that if we want to show that an LTS $\mathcal{T}$ satisfies a property p, then slicing that system *w.r.t.* $\mathsf{dep}_p\ \mathcal{T}$ will not change the system's ability to satisfy p. We do not formally prove this, but we refer to Hatcliff et al. (2000) for a proof that properties (expressed in LTL) are preserved by slicing. Below, we will show the formalisation of this assumption.

The following definitions show how we slice states, LTSs and traces. For all slicing operations, we fix a set of variables V on which the property ϕ depends. Remember that states are defined as a mapping from variables to values; a sliced state is a restriction of this mapping to the variables in V. Since all functions in Isabelle have to be total, we map all other variables to some unknown constant arbitrary.

Definition 7 (Sliced state) *Given a state s, we define the* sliced state $s_{|V}$ *as follows:*

$$s_{|V} = \lambda v.\ \text{if } v \in V \text{ then } s(v) \text{ else arbitrary}$$

Notice that we immediately can prove the following results for sliced states:

$$\begin{array}{rcl} s =_V t & \Leftrightarrow & s_{|V} = t_{|V} \\ (s_{|V})_{|V'} & = & s_{|V \cap V'} \\ V \subseteq V' & \Rightarrow & (s_{|V'})_{|V} = s_{|V} \\ & & s_{|V} =_V s \end{array}$$

[5] In many cases, such as for the rule for Exists p Between $q\,r$, it suffices to show that there is an equivalent initial fragment.

For convenience, given an arbitrary boolean state predicate C mentioning only variables in V, we use $C_{|V}$ to denote the sliced state predicate

$$\lambda s.\ \exists s'.\ C(s') \wedge (s'_{|V} = s)$$

Next, we define a sliced LTS. The sliced transition relation is defined as a restriction of the original transition relation, only keeping the transitions that affect variables in V. Note that initial states are preserved by slicing.

Definition 8 (Sliced LTS) *Given LTS $\mathcal{T} = (S, A, \rightarrow, I)$, we define a* sliced LTS w.r.t. *the set V, as* $\mathsf{slice}\ \mathcal{T}\ V = (S', A', \rightarrow', I')$, *where*

- $S' = \bigcup_{s \in S} s_{|V}$
- $A' = A$
- $\rightarrow' = \{(s', a, t') \mid \exists s\ t.\ s \xrightarrow{a} t \wedge s_{|V} = s' \wedge t_{|V} = t' \wedge s' \neq t'\}$
- $I' = \bigcup_{i \in I} i_{|V}$

As mentioned above, an important assumption of our model is the slicing assumption which states that properties are preserved by slicing, *i.e.* an LTS $\mathcal{T}$ q-satisfies a temporal property *iff* $\mathsf{slice}\ \mathcal{T}\ V$ q-satisfies the same property. We formalise this assumption as follows:

$$\mathcal{T},\ q \models \phi \Leftrightarrow \mathsf{slice}\ \mathcal{T}\ V,\ q_{|V} \models \phi \qquad \text{(Ass 6)}$$

Notice that q can be instantiated with $\lambda s.\ s \in I$, when using this assumption for temporal properties with scope Before or Globally.

In order to define sliced traces, we use an auxiliary function nrss (for Number of Sliced States), which counts the number of different states (*w.r.t. V*) in the first k states of a trace. This function is recursively defined by the following two equations:

$$\begin{array}{l}\mathsf{nrss}\ V\ x\ 0 = 0\\ \mathsf{nrss}\ V\ x\ (\mathsf{Suc}\ k) = \\ \qquad (\mathsf{if}\ x_k =_V x_{\mathsf{Suc}\ k}\ \mathsf{then}\ 0\ \mathsf{else}\ 1) + \\ \qquad \mathsf{nrss}\ V\ x\ k\end{array}$$

Notice that this function is monotonous in its last argument.

$$i \leq j \Rightarrow \mathsf{nrss}\ V\ x\ i \leq \mathsf{nrss}\ V\ x\ j$$

In addition, if the number of sliced states up to i is the same as the number of sliced states up to j, this implies that x_i is V-equivalent to x_j.

$$\mathsf{nrss}\ V\ x\ i = \mathsf{nrss}\ V\ x\ j \Rightarrow x_i =_V x_j$$

Notice that the converse is not necessarily the case, since an LTS can reach a single state several times.

Finally, we are ready to define a sliced trace.

Definition 9 (Sliced trace) *Given a trace x, we define a* sliced trace *w.r.t. V as follows:*

$$\begin{array}{l}\mathsf{slice\_trace}\ V\ x = \\ \qquad \lambda i.\ \mathsf{if}\ (\exists j.\ i \leq \mathsf{nrss}\ V\ x\ j) \\ \qquad\quad \mathsf{then}\ (x_{(\mathsf{least}\ j. \mathsf{nrss}\ V\ x\ j = i)})_{|V} \\ \qquad\quad \mathsf{else}\ (x_{(\mathsf{least}\ j.\ \forall k.\ j \leq k \Rightarrow \mathsf{nrss}\ V\ x\ j = \mathsf{nrss}\ V\ x\ k)})_{|V}\end{array}$$

If we want to know what the i^{th} state is in $\mathsf{slice\_trace}\ V\ x$, we do the following.

- First we check whether there exists at least i different states in x, *w.r.t.* V.
- If this is the case, then if j is the smallest number for which $\mathsf{nrss}\ V\ x\ j = i$, then we return x_j restricted to V.
- Otherwise, there are less than i different states, thus the sliced trace is stuttering.
- Suppose x_j is the first state where the stuttering begins, *i.e.* afterwards the number of sliced states remains constant: $\forall k. j \leq k \Rightarrow \mathsf{nrss}\ V\ x\ j = \mathsf{nrss}\ V\ x\ k$. In that case we return x_j restricted to V.

To conclude, we present two properties concerning sliced traces. The first property shows how the sliced trace relates to the function nrss: for any i, if j is the number of different sliced states up to i, then the j^{th} state in the sliced trace is equal to x_i restricted to V.

$$(\mathsf{slice\_trace}\ V\ x)_{\mathsf{nrss}\ V\ x\ i} = (x_i)_{|V}$$

The second property tells us that every state in the sliced trace can be related to a state in the original trace, *i.e.* for the i^{th} state in the sliced trace there exists a j such that this state is equal to x_j, restricted to V, and either the number of different sliced states up to j is equal to i, or the number of different sliced states is strictly less than i for any k. In the latter case, the sliced trace is stuttering from the i^{th} state onwards.

$$\begin{array}{l}\exists j.\ (\mathsf{slice\_trace}\ V\ x)_i = (x_j)_{|V} \wedge \\ \qquad\qquad (\mathsf{nrss}\ V\ x\ j = i \vee \\ \qquad\qquad\ \forall k.\ \mathsf{nrss}\ V\ x\ k < i)\end{array} \qquad (3)$$

Verification of the rule for Exists p Between $q\ r$. Next, we shall look at the proof for the rule for the formula Exists p Between $q\ r$ in detail. Below, we write ϕ for Exists p Between $q\ r$.

$$\frac{\mathcal{T}_1,\ q \models \phi \quad \neg p \wedge \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ |\ p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \phi}$$

First we observe that because of equation (2) it is sufficient to show that $\mathcal{T}_1 \| \mathcal{T}_2$ q-satisfies this property. Second, we observe that the pattern Exists p Between $q\ r$ maps into the LTL formula

$$[\,]\,((q \wedge \neg r) \Rightarrow (\neg r \mathsf{W} (p \wedge \neg r)))$$

Using the semantics of LTL, this tells us that we have to show the following:

$$\begin{array}{l}\forall x.\ \mathsf{trace\_q}\ q\ (\mathcal{T}_1 \| \mathcal{T}_2)\ x \Rightarrow \\ \qquad \forall i.\ q(x_i)\ \wedge\ \neg r(x_i) \Rightarrow \\ \qquad\quad (\exists k.\ p(x_{i+k})\ \wedge\ \neg r(x_{i+k}) \wedge \\ \qquad\qquad\ (\forall j.\ j < k \Rightarrow \neg r(x_{i+j}))) \vee \\ \qquad\quad (\forall l.\ \neg r(x_{i+l})))\end{array}$$

Suppose we have x and i such that $\mathsf{trace\_q}\ q\ \mathcal{T}_1 \| \mathcal{T}_2\ x$, $q(x_i)$ and $\neg r(x_i)$. Notice that we immediately have $\mathsf{trace\_q}\ q\ \mathcal{T}_1 \| \mathcal{T}_2\ (x^i)$.

We wish to prove by *absurdum*, *i.e.* we assume the negation of the conclusion and we try to establish a contradiction. Assuming the negation of the conclusion gives us the following extra assumptions:

- $\forall k. p(x_{i+k})\ \Rightarrow\ (r(x_{i+k})\ \vee\ \exists j.\ j < k \wedge r(x_{i+j}))$
- $\exists l.\ r(x_{i+l})$

Let l be given such that r is true in the $i + l^{th}$ state of x; we know that there must be a smallest n such that r is true.

$$r(x_{i+n}) \wedge \forall j.\ r(x_{i+j}) \Rightarrow n \leq j$$

Using the assumptions, we can then easily derive the following property:

$$\forall j.\ j < n \Rightarrow \neg p(x_{i+j}) \tag{4}$$

Next we apply the slicing assumption (Ass 6) to the first hypothesis of the rule which gives us (remember $\phi =$ Exists p Between $q\ r$)

$$(\mathsf{slice}\ \mathcal{T}_1\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)),\ q_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)} \models \phi \tag{5}$$

To be able to use this result, we show that given the different hypotheses, if x^i is a q-trace of the composed LTS and n is the smallest number such that $r(x_{i+n})$ holds, then slicing x^i *w.r.t.* $\mathsf{dep}_{p,r}\ \mathcal{T}_1$ returns an initial trace fragment of $\mathsf{slice}\ \mathcal{T}_1\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)$. This initial trace fragment takes at least $\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ n$ steps of the sliced LTS. Formally:

$$\begin{array}{l}
\mathcal{T}_1, q \models \mathsf{Exists}\ p\ \mathsf{Between}\ q\ r \Rightarrow \\
\neg p\ \wedge\ \neg r \models \mathcal{T}_2\ \mathsf{preserves}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1) \mid p \Rightarrow \\
\mathsf{trace_q}\ q\ \mathcal{T}_1 \| \mathcal{T}_2\ (x^i) \Rightarrow \\
r(x_{i+n})\ \Rightarrow \\
\forall j.\ r(x_{i+j}) \Rightarrow n \leq j \Rightarrow \\
\quad \mathsf{trace_q_upto}\ \ q_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)} \\
\qquad\qquad\qquad\ \ \mathsf{slice}\ \mathcal{T}_1\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1) \\
\qquad\qquad\qquad\ \ \mathsf{slice_trace}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i) \\
\qquad\qquad\qquad\ \ \mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ n
\end{array}$$

Using equation (1), we know that this initial trace fragment can be extended to a trace of $\mathsf{slice}\ \mathcal{T}_1\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)$.

Our next step is to let the extended trace starting with $\mathsf{slice\_trace}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)$ be called y. We can derive immediately that $q_{|\mathsf{dep}_{p,r}\ \mathcal{T}}(y_0)$. By using (5), we can immediately conclude that y satisfies the temporal property Exists p Between $q\ r$. If we use the mapping into LTL and the LTL semantics, we can instantiate the resulting formula with $i = 0$. This tells us that we have

$$\begin{array}{l}
(\exists k.\ p_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_k) \wedge \\
\qquad \neg r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_k)\ \wedge \\
\qquad \forall j.j < k \Rightarrow \neg r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_j)) \\
\vee \\
(\forall l.\ \neg r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_l))
\end{array} \tag{6}$$

Notice that using (Ass 5) and the definition of V-equality and sliced states, we get for all k and l

$$\begin{array}{l}
(x_k)_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)} = y_l \Rightarrow p(x_k) \Leftrightarrow p_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_l) \\
(x_k)_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)} = y_l \Rightarrow r(x_k) \Leftrightarrow r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_l)
\end{array} \tag{7}$$

We use these properties and a case distinction on the disjunction in (6) above to finish the proof.

Case 1: first disjunct
Suppose there exists a k such that $p_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_k)$, $\neg r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_k)$ and $\forall j.\ j < k \Rightarrow \neg r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_j)$. We begin by making a case distinction on whether $\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ n$ is less than k.

Case 1.1: $\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ n < k$
In this particular case, predicate r does not hold in the state $y_{\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ n}$. However this state is equivalent to $(x_{i+n})_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}$ and we already know that $r(x_{i+n})$, which gives a contradiction.

Case 1.2: $\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ n \geq k$
By using equation (3) we know that we can relate y_k to a state x_m such that either $\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ m = k$ or $\forall j.\ \mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ j < k$. We find that this latter inequality immediately gives us a contradiction with $\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ n \geq k$, therefore

$$\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ m = k$$

Next, we make a case distinction on whether m is smaller than n.

Case 1.2.1: $m < n$
We know by (4) that $\neg p(x_{i+m})$, thus $\neg p_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_k)$, which immediately creates a contradiction with the assumption $p_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_k)$.

Case 1.2.2: $n \leq m$
Using monotonicity of the function nrss, we derive the equivalence

$$\mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ n = \mathsf{nrss}\ (\mathsf{dep}_{p,r}\ \mathcal{T}_1)\ (x^i)\ m$$

Thus $x_{i+n} =_{(\mathsf{dep}_{p,r}\ \mathcal{T}_1)} x_{i+m}$, and therefore $r(x_{i+m})$ and thus $r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_k)$, which also leads to a contradiction.

Case 2: second disjunct
First we suppose that $\forall l.\ \neg r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_l)$. We had assumed that $r(x_{i+l})$. There exists a state in y that is the restriction of x_{i+l} to $\mathsf{dep}_{p,r}\ \mathcal{T}_1$. By property (7) above, $r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}$ holds in this state, which immediately leads to a contradiction with $\forall l.\ \neg r_{|(\mathsf{dep}_{p,r}\ \mathcal{T}_1)}(y_l)$.

This concludes the proof.

6 Example

To illustrate how our factorisation method works in practice, we consider the code fragment in Figure 2 (adapted from Corbett et al. (2001)). This fragment defines a class `Buffer`, that is accessed by three different threads, a consumer $\mathcal{C}$ and a producer $\mathcal{P}$ and a thread that processes the buffer by moving the items in the incoming buffer to the outcoming buffer $\mathcal{PB}$.

Typical properties that one may wish to verify are for example:

- if the incoming buffer is full, it eventually will become non-full; and
- after the incoming buffer has become non-empty, eventually the outgoing buffer also will become non-empty.

Using the specification patterns, we can specify these properties formally.

$$\begin{array}{ll}
(\phi) & \texttt{!Buffer.inIsFull()}\ \mathsf{RespondsTo} \\
& \qquad \texttt{Buffer.inIsfull()}\ \mathsf{Globally} \\
(\psi) & \mathsf{Exists}\ \texttt{!Buffer.outIsEmpty()} \\
& \qquad \mathsf{After}\ \texttt{!Buffer.inIsEmpty()}
\end{array}$$

Using our factorisation rules, we can show that it is sufficient to prove that these properties are guaranteed by the ProcessBuffer thread $\mathcal{PB}$ only. Notice that we only have to show the property ψ for traces where initially the incoming buffer is not empty. We can use any existing techniques for the verification of (multi-threaded) programs to establish this.

```
final class Buffer{
 int [] inbuf, outbuf;
 int inbound, outbound, inhead, outhead,
     intail, outtail;

 public Buffer(int inb, int outb){
  inbound = inb; outbound = outb;
  inbuf = new int[inbound];
  outbuf = new int[outbound];
  inhead = 0; outhead = 0;
  intail = inbound - 1; outtail = outbound - 1;}

 public synchronized boolean inIsFull(){
  return inhead == intail;}

 public synchronized boolean outIsFull(){
  return outhead == outtail;}

 public synchronized boolean inIsEmpty(){
  return inhead == ((intail+1)%inbound);}

 public synchronized boolean outIsEmpty(){
  return outhead == ((outtail+1)%outbound);}

 public synchronized void add(int o) {
  while (inIsFull())
    try {wait();}catch(InterruptedException e){};
  inbuf[inhead] = o;
  inhead = (inhead+1)% inbound;
  notifyAll();}

public synchronized void process() {
  while (inIsEmpty())
    try {wait();}catch(InterruptedException e){}
  intail = (intail + 1) % inbound;
  while (outIsFull())
    try {wait();}catch(InterruptedException e){}
  intail = (intail+1) % inbound;
  outbuf[outhead] = inbuf[intail];
  outhead = (outhead+1)%outbound;
  notifyAll();}

public synchronized int take() {
  while (outIsEmpty())
    try {wait();}catch(InterruptedException e){}
  outtail = (outtail+1)%outbound;
  notifyAll();
  return outbuf[outtail];}
}

final class ProcessBuffer extends Thread{
  Buffer buf;
  public ProcessBuffer(Buffer b) {buf = b;}
  public void run(){
        while(true)buf.process();}
}

final class Producer extends Thread{
  Buffer buf;
  public Producer(Buffer b) {buf = b;}
  public void run(){
    int i = 0;
    while(true){buf.add(i); i++;}}
}

final class Consumer extends Thread{
  Buffer buf;
  public Consumer(Buffer b) {buf = b;}
  public void run(){
    while(true)System.out.println(buf.take());}
}
```

Figure 2: A buffer and three parallel threads: `Consumer`, `Producer` and `ProcessBuffer`

$$\frac{\mathcal{PB} \models \phi}{\mathcal{PB}, !\texttt{Buffer.inIsEmpty}() \models \psi} \tag{8}$$

In addition, we need to show that the producer $\mathcal{P}$ and consumer $\mathcal{C}$ do not disturb the validity of the properties provided the appropriate conditions hold, *i.e.* provided the incoming buffer is full, respectively, the outgoing buffer is empty. In order to do this, we first need to determine the appropriate dependency sets. Using a standard dependency analysis, we find the following sets:

$$\begin{array}{rcl} \mathsf{dep}_{\phi}\,\mathcal{PB} & = & \{\texttt{inhead}, \texttt{intail}, \texttt{inbound}\} \\ \mathsf{dep}_{\psi}\,\mathcal{PB} & = & \{\texttt{outhead}, \texttt{outtail}, \texttt{outbound}\} \end{array}$$

Now, appropriately instantiating the factorisation rules for RespondsTo Globally and Exists After, we find that we have the following extra proof obligations:

$$\begin{array}{l} \texttt{Buffer.inIsFull}() \models \\ \quad \mathcal{C} \| \mathcal{P} \text{ preserves } \mathsf{dep}_{\phi}\,\mathcal{PB} \mid !\texttt{Buffer.inIsFull}() \\ \texttt{Buffer.outIsEmpty}() \models \\ \quad \mathcal{C} \| \mathcal{P} \text{ preserves } \mathsf{dep}_{\psi}\,\mathcal{PB} \mid !\texttt{Buffer.outIsEmpty}() \end{array} \tag{9}$$

It is straightforward to see that these proof obligations are satisfied. However, notice that without the extra conditions, these factorisations would not have been possible. We explicitly use that the producer $\mathcal{P}$ does not produce any new elements when the incoming buffer is full, and *vice versa*; that the consumer $\mathcal{C}$ does not take any elements when the outgoing buffer is empty.

From (8) and (9) we can conclude the following:

$$\begin{array}{l} \mathcal{PB} \| \mathcal{C} \| \mathcal{P} \models \phi \\ \mathcal{PB} \| \mathcal{C} \| \mathcal{P} \models \psi \end{array}$$

Notice that if these three threads had been used in a larger context many of the typical other properties on buffers (*e.g.* all elements that are taken first must have been added) can be factorised to these three threads only; for all other threads one only would have to show that they do not disturb the validity of the property.

Using this buffer example, it is worth comparing our approach to the three discussed in the introduction: slicing, abstraction and atomicity checking. Recall that slicing techniques aid verification by removing instructions irrelevant to a property being checked. For our example, slicing techniques would be of no benefit since all three classes use the same variables. Abstraction techniques involve "abstracting" a program P into a smaller, more manageable transition system representing P. In our case this would not be particularly worthwhile, since the level of the property is close to the level of the application. Atomicity checking determines for each method whether the interleavings of its instructions gives the same result as executing its instructions without interleavings. Because we are more interested in verifying sequences of methods/actions, rather than conducting verification on a method *per* method basis, atomicity checking is not relevant in this case.

7 Conclusions

We have presented a method to factorise the verification of temporal properties for multi-threaded programs over different threads. Contrary to other approaches that aim to reduce the verification burden by eliminating unnecessary verification tasks for the entire application, our approach is more modular in nature. We decompose the program into different parts for which different verification tasks exist, giving added flexibility to program verification. With the growing popularity of component-based software, *e.g.* JavaBeans, such flexibility is sorely needed. We feel that

our technique can be used to improve the applicability of other, non-modular techniques.

As a property specification language we have used the specification patterns developed within the Bandera project. This language, along with our program model, has been formalised in Isabelle/HOL. We have designed 25 rules that describe the factorisation of a given temporal property and have proven each rule correct *w.r.t.* to our formalisation. We have also identified and corrected minor deficiencies within the patterns.

As future work, we would like to develop an automatic technique to check for the preservation of variables. We believe that it will be possible to define this as an extension of existing techniques for checking so-called frame conditions (Spoto & Poll 2003, Cataño & Huisman 2003), *i.e.* specification clauses that describe which variables may be modified by a method.

A natural extension to our approach will be to take invariants into account (following Prasetya & Swierstra (2003)). If a property J is known to hold in all reachable program states, *i.e.* if it is an invariant, then this can be used to ease the verification process. The factorisation rules could be changed as follows:

$$\frac{\mathcal{T}_1, J \models \phi \qquad C_1, J \models \mathcal{T}_2 \text{ preserves } (\mathsf{dep}_{C_2}\ \mathcal{T}_1) \mid C_3}{\mathcal{T}_1 \| \mathcal{T}_2, J \models \phi}$$

Intuitively, the proof rule would now read: in order to prove that the composed system $\mathcal{T}_1 \| \mathcal{T}_2$ satisfies property ϕ, assuming that we have an invariant J, it is sufficient to prove that (1) $\mathcal{T}_1$ satisfies ϕ assuming the invariant J, and (2) $\mathcal{T}_2$ preserves the variables on which V depends, also assuming the invariant J. One could also imagine using other, more elaborate properties as additional assumptions to the factorisation. It is the subject of future work to study those kind of properties which would be useful.

References

Breunesse, C., Cataño, N., Huisman, M. & Jacobs, B. (2003), Formal methods for smart cards: an experience report, Technical Report NIII-R0316, NIII, University of Nijmegen. To appear in *Science of Computer Programming*.

Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K. & Poll, E. (2003), An overview of JML tools and applications, *in* T. Arts & W. Fokkink, eds, 'Formal Methods for Industrial Critical Systems (FMICS 2003)', Vol. 80 of *ENTCS*, Elsevier.

Cataño, N. & Huisman, M. (2003), Chase: a static checker for JML's assignable clause, *in* L. Zuck, P. Attie, A. Cortesi & S. Mukhopadhyay, eds, 'Verification, Model Checking and Abstract Interpretation', number 2575 *in* 'LNCS', Springer, pp. 26–40.

Cenciarelli, P., Knapp, A., Reus, B. & Wirsing, M. (1999), An event-based structural operational semantics of multi-threaded Java, *in* J. Alves-Foss, ed., 'Formal Syntax and Semantics of Java', number 1523 *in* 'LNCS', Springer, pp. 175–200.

Clarke, E., Grumberg, O. & Long, D. (1994), 'Model checking and abstraction', *ACM Transactions on Programming Languages and Systems* **16**(5), 1512–1542.

Clarke, E., Long, D. & McMillan, K. (1989), Compositional model checking, *in* 'Fourth IEEE Symposium on Logic in Computer Science', IEEE Press, pp. 353–362.

Corbett, J., Dwyer, M., Hatcliff, J. & Robby (2001), Expressing Checkable Properties of Dynamic Systems: the Bandera Specification Language, Technical Report 2001-04, Kansas State University, Department of Computing and Information Sciences.

Dwyer, M., Avrunin, G. & Corbett, J. (1998), Property specification patterns for finite-state verification, *in* M. Ardis, ed., '2nd Workshop on Formal Methods in Software Practice (FMSP'98)', pp. 7–15.

Emerson, E. (1990), *Temporal and Modal Logic*, Elsevier, chapter 16.

Flanagan, C. & Freund, S. (2004), Atomizer: A dynamic atomicity checker for multithreaded programs, *in* X. Leroy, ed., 'Principles of Programming Languages (POPL 2004)', pp. 256–267.

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2000), *The Java Language Specification, second edition*, The Java Series, Addison-Wesley.

Hatcliff, J., Corbett, J., Dwyer, M., Sokolowski, S. & Zheng, H. (1999), A formal study of slicing for multi-threaded programs with JVM concurrency primitives, *in* 'International Symposium on Static Analysis (SAS'99)', number 1694 *in* 'LNCS', Springer, pp. 1–18.

Hatcliff, J., Dwyer, M. & Zheng, H. (2000), 'Slicing software for model construction', *Higher-Order and Symbolic Computation* **13**(4), 315–353.

Hatcliff, J., Robby & Dwyer, M. (2004), Verifying atomicity specifications for concurrent object-oriented software using model checking, *in* B. Steffen & G. Levi, eds, 'Verification, Model Checking and Abstract Interpretation (VMCAI 2004)', number 2937 *in* 'LNCS', Springer, pp. 175–190.

Laster, K. & Grumberg, O. (1998), Modular model checking of software, *in* 'Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)', Vol. 1384 of *LNCS*, Springer, pp. 20–35.

Mateescu, R. (1998), Local model-checking of an alternation-free value-based modal mu-calculus, *in* A. Bossi, A. Cortesi & F. Levi, eds, '2nd International Workshop on Verification, Model Checking and Abstract Interpretation'.

Nipkow, T., Paulson, L. & Wenzel, M. (2002), *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, number 2283 *in* 'LNCS', Springer.

Prasetya, I. & Swierstra, S. (2003), 'Factorizing fault tolerance', *Theor. Comp. Sci.* **290(2)**, 1201–1222.

Robby, Rodríguez, E., Dwyer, M. & Hatcliff, J. (2004), Checking strong specifications using an extensible software model checking framework, *in* 'Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)', pp. 404–420.

Santone, A. (2002), 'Automatic verification of concurrent systems using a formula-based compositional approach', *Acta Informatica* **38**, 531–564.

Spoto, F. & Poll, E. (2003), Static analysis for JML's assignable clauses, *in* G. Ghelli, ed., 'Foundations of Object-Oriented Languages (FOOL-10)'.

Sprenger, C., Gurov, D. & Huisman, M. (2004), Compositional verification for secure loading of smart card applets, *in* C. Heitmeyer & J.-P. Talpin, eds, 'Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'04)', IEEE, pp. 211–222.

A Universality Rules

$$\alpha = \text{Universal } p \text{ Globally}$$
$$\beta = \text{Universal } p \text{ After } q$$
$$\gamma = \text{Universal } p \text{ Before } r$$
$$\delta = \text{Universal } p \text{ Between } q\ r$$
$$\epsilon = \text{Universal } p \text{ AfterUntil } q\ r$$
$$V = \text{dep}_{p,r}\ \mathcal{T}_1$$

$$\frac{\mathcal{T}_1 \models \alpha \quad p \models \mathcal{T}_2 \text{ preserves } (\text{dep}_p\ \mathcal{T}_1) \mid \text{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \alpha}$$

$$\frac{\mathcal{T}_1,\ q \models \beta \quad p \models \mathcal{T}_2 \text{ preserves } (\text{dep}_p\ \mathcal{T}_1) \mid \text{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \beta}$$

$$\frac{\mathcal{T}_1 \models \gamma \quad \neg r \models \mathcal{T}_2 \text{ preserves } V \mid \text{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \gamma}$$

$$\frac{\mathcal{T}_1,\ q \models \delta \quad \neg r \models \mathcal{T}_2 \text{ preserves } V \mid \text{false}}{\mathcal{T}_1 \| \mathcal{T}_2 \models \delta}$$

$$\frac{\mathcal{T}_1,\ q \models \epsilon \quad p \wedge \neg r \models \mathcal{T}_2 \text{ preserves } V \mid r}{\mathcal{T}_1 \| \mathcal{T}_2 \models \epsilon}$$

B Existence Rules

$$\alpha = \text{Exists } p \text{ Globally}$$
$$\beta = \text{Exists } p \text{ After } q$$
$$\gamma = \text{Exists } p \text{ Before } r$$
$$\delta = \text{Exists } p \text{ Between } q\ r$$
$$\epsilon = \text{Exists } p \text{ AfterUntil } q\ r$$
$$V = \text{dep}_{p,r}\ \mathcal{T}_1$$

$$\frac{\mathcal{T}_1 \models \alpha \quad \neg p \models \mathcal{T}_2 \text{ preserves } \text{dep}_p\ \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \alpha}$$

$$\frac{\mathcal{T}_1,\ q \models \beta \quad \neg p \models \mathcal{T}_2 \text{ preserves } \text{dep}_p\ \mathcal{T}_1 \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \beta}$$

$$\frac{\mathcal{T}_1 \models \gamma \quad \neg p \wedge \neg r \models \mathcal{T}_2 \text{ preserves } V \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \gamma}$$

$$\frac{\mathcal{T}_1,\ q \models \delta \quad \neg p \wedge \neg r \models \mathcal{T}_2 \text{ preserves } V \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \delta}$$

$$\frac{\mathcal{T}_1,\ q \models \epsilon \quad \neg p \wedge \neg r \models \mathcal{T}_2 \text{ preserves } V \mid p \vee r}{\mathcal{T}_1 \| \mathcal{T}_2 \models \epsilon}$$

C Precedence Rules

$$\alpha = p \text{ Precedes } q \text{ Globally}$$
$$\beta = p \text{ Precedes } q \text{ After } r$$
$$\gamma = p \text{ Precedes } q \text{ Before } r$$
$$\delta = p \text{ Precedes } q \text{ Between } q\ r$$
$$\epsilon = p \text{ Precedes } q \text{ AfterUntil } q\ r$$
$$V = \text{dep}_{p,q}\ \mathcal{T}_1$$
$$W = \text{dep}_{p,q,r}\ \mathcal{T}_1$$
$$X = \text{dep}_{p,q,s}\ \mathcal{T}_1$$

$$\frac{\mathcal{T}_1 \models \alpha \quad \neg p \wedge \neg q \models \mathcal{T}_2 \text{ preserves } V \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \alpha}$$

$$\frac{\mathcal{T}_1,\ r \models \beta \quad \neg p \wedge \neg q \models \mathcal{T}_2 \text{ preserves } V \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \beta}$$

$$\frac{\mathcal{T}_1 \models \gamma \quad \neg p \wedge \neg q \wedge \neg r \models \mathcal{T}_2 \text{ preserves } W \mid p \vee r}{\mathcal{T}_1 \| \mathcal{T}_2 \models \gamma}$$

$$\frac{\mathcal{T}_1,\ r \models \delta \quad \neg p \wedge \neg q \wedge \neg s \models \mathcal{T}_2 \text{ preserves } X \mid p \vee s}{\mathcal{T}_1 \| \mathcal{T}_2 \models \delta}$$

$$\frac{\mathcal{T}_1,\ r \models \epsilon \quad \neg p \wedge \neg q \wedge \neg s \models \mathcal{T}_2 \text{ preserves } X \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \epsilon}$$

D Response Rules

$$\alpha = p \text{ RespondsTo } q \text{ Globally}$$
$$\beta = p \text{ RespondsTo } q \text{ After } r$$
$$\gamma = p \text{ RespondsTo } q \text{ Before } r$$
$$\delta = p \text{ RespondsTo } q \text{ Between } r\ s$$
$$\epsilon = p \text{ RespondsTo } q \text{ AfterUntil } r\ s$$
$$V = \text{dep}_{p,q}\ \mathcal{T}_1$$
$$W = \text{dep}_{p,q,r}\ \mathcal{T}_1$$
$$X = \text{dep}_{p,q,s}\ \mathcal{T}_1$$

$$\frac{\mathcal{T}_1 \models \alpha \quad \neg p \models \mathcal{T}_2 \text{ preserves } V \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \alpha}$$

$$\frac{\mathcal{T}_1,\ r \models \beta \quad \neg p \models \mathcal{T}_2 \text{ preserves } V \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \beta}$$

$$\frac{\mathcal{T}_1 \models \gamma \quad \neg p \wedge \neg r \models \mathcal{T}_2 \text{ preserves } W \mid p \vee r}{\mathcal{T}_1 \| \mathcal{T}_2 \models \gamma}$$

$$\frac{\mathcal{T}_1,\ r \models \delta \quad \neg p \wedge \neg s \models \mathcal{T}_2 \text{ preserves } X \mid p \vee s}{\mathcal{T}_1 \| \mathcal{T}_2 \models \delta}$$

$$\frac{\mathcal{T}_1,\ r \models \epsilon \quad \neg p \wedge \neg s \models \mathcal{T}_2 \text{ preserves } X \mid p}{\mathcal{T}_1 \| \mathcal{T}_2 \models \epsilon}$$