Toward Static Timing Analysis of Parallel Software – Technical Report*

Andreas Gustavsson, Jan Gustafsson, and Björn Lisper

School of Innovation Design and Engineering, Mälardalen University, Sweden
{andreas.sg.gustavsson,jan.gustafsson,bjorn.lisper}@mdh.se

Abstract

The current trend within computer, and even real-time, systems is to incorporate parallel hardware, e.g., multicore processors, and parallel software. Thus, the ability to safely analyse such parallel systems, e.g., regarding the timing behaviour, becomes necessary. Static timing analysis is an approach to mathematically derive *safe* bounds on the execution time of a program, when executed on a given hardware platform. This paper presents an algorithm that statically analyses the timing of parallel software, with threads communicating through shared memory, using abstract interpretation. It also gives an extensive example to clarify how the algorithm works.

Note that the contents of this report might be updated and/or completed without public notification.

1998 ACM Subject Classification F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages–*Program Analysis*

Keywords and phrases Parallelism, BCET, WCET, Static analysis, Abstract interpretation

1 Introduction

Many safety-critical embedded systems have hard real-time requirements. For these, safe bounds on the Best- and Worst-Case Execution Times (BCET/WCET) of the tasks in the system are key measures. Together, they define an interval in time within which the execution of the task is guaranteed to finish. In particular WCET bounds are needed by, e.g., schedulability analyses.

For reasons of energy consumption and performance, development in hardware today strives toward massively parallel architectures, like many-core, GPU and even special purpose, heterogeneous platforms. Thus, it is very likely that software tasks in future real-time systems will be parallel in order to utilise the provided computing power. Therefore, efforts must be made in providing WCET analyses for such systems.

This paper focuses on analysing the timing behaviour of *parallel software* with *dependent* sub-tasks, using a programming model with threads, shared memory, and locks. This kind of programming model is commonly used in parallel software today. It is assumed that an arbitrary underlying timing model, which can safely predict safe bounds on the BCET and WCET of individual instructions given a certain system state, is provided. An algorithm to statically derive the BCET and WCET of parallel software using abstract interpretation is presented.

The rest of this paper is organised as follows. Section 2 presents related work on static timing analysis for parallel systems. Section 3 gives a short introduction to abstract

* This work was partly funded by the Swedish Research Council (VR) through project 2008-4650, "Worst-Case Execution Time Analysis of Parallel Systems".

Technical Report number 2796, published at http://www.mrtc.mdh.se/index.php?choice=publications&id=2796.

interpretation. Section 4 introduces a small model parallel language, with threads, thread-local and global memory, and locks. We also give a formal semantics for the language, including time, and we then present the analysis. Section 5 clarifies how the analysis works by instantiating it for a given example program. Section 6 concludes the presentation with some discussion and directions for the future.

2 Related Work

As far as we know, there have not been many attempts to statically analyse the execution time of explicitly parallel software. The MERASA project provides a timing analysable multicore CPU with a system level software (c.f., operating system). In [11], a case study is performed, in which the WCET of a parallel 3D multigrid solver, executing on the MERASA platform, is derived. In [8], model-checking (not a static approach) is used to derive the WCET of a minimal parallel program. It is shown that, since model-checking is based on exhaustive exploration of concrete states, it is difficult to achieve scalability using only the presented approach. In [9], abstract interpretation is combined with model-checking to avoid the scalability problems found in, e.g., [8]. This work does not focus on explicitly parallel software, though.

In [3], an approach to directly calculate the BCET and WCET for sequential programs using abstract execution [7] is presented. Our work takes basically the same approach, but for explicitly parallel programs.

There is also some research on static low-level analysis of parallel systems. In [1] and [12], static methods for analysing multicores with a shared L2 instruction cache are presented. In [1], effects from timing anomaly influenced pipelines are also taken into account.

3 Background Theory

In general, basing a timing analysis on the concrete semantic of a program is infeasible due to the enormous number of states that must be explored. Abstract interpretation [2, 5, 10] is a method for *safely* approximating the concrete program semantics and can be used to obtain a set of possible abstract states for each point in a program. An abstract state collects, and most often over-approximates, the information given by a *set* of concrete semantic states. This means that an analysis based on abstractly interpreting the semantics of a program can become less complex and more efficient, but might suffer from imprecision, compared to an analysis based on the concrete semantics.

3.1 Galois Connections

The concrete semantics of a programming language can be abstracted in many different ways. The choice of abstraction is done by defining an abstract domain. An abstract domain is essentially the set of all possible abstract states that fit the definition of the domain. It is often shown that the abstract domain is a safe over-approximation of the concrete domain by deriving a Galois connection between the two domains [10]. A Galois connection between two domains (complete lattices[1]), D and $\tilde{D}$, is described by an abstraction function, α, and a concretisation function, γ, which must fulfil the criterion in Definition 3.1.

[1] An introduction to the theory on complete lattices can be found in many textbooks, e.g., [10].

▶ **Definition 3.1** (Galois connection). $\langle \alpha, \gamma \rangle$ is a Galois connection iff α and γ are monotone functions that fulfil

$$\alpha \circ \gamma \sqsubseteq \lambda \tilde{d}.\tilde{d}$$
$$\gamma \circ \alpha \sqsupseteq \lambda d.d$$

for all $d \in D$ and $\tilde{d} \in \tilde{D}$, where D is the concrete domain and $\tilde{D}$ is the abstract domain. ◀

An often useful special case of a Galois connection is called a Galois insertion [10].

▶ **Definition 3.2** (Galois insertion). $\langle \alpha, \gamma \rangle$ is a Galois insertion iff α and γ are monotone functions that fulfil

$$\alpha \circ \gamma = \lambda \tilde{d}.\tilde{d}$$
$$\gamma \circ \alpha \sqsupseteq \lambda d.d$$

for all $d \in D$ and $\tilde{d} \in \tilde{D}$, where D is the concrete domain and $\tilde{D}$ is the abstract domain. ◀

A function in the concrete domain, $f : D \to D$, can be safely approximated by a function in the abstract domain, $\tilde{f} : \tilde{D} \to \tilde{D}$, iff $\forall \tilde{d} \in \tilde{D} : f(\gamma(\tilde{d})) \sqsubseteq \gamma(\tilde{f}(\tilde{d}))$. The best approximation is achieved by inducing f along α [10].

▶ **Definition 3.3** (Induced function). Assuming that $\langle \alpha, \gamma \rangle$ is a Galois connection, the best approximation of $f : D \to D$ in $\tilde{D} \to \tilde{D}$ is given by:

$$\alpha \circ f \circ \gamma$$ ◀

Sometimes, it is more convenient to work with adjunctions instead of Galois connections.

▶ **Definition 3.4** (Adjunction). $\langle \alpha : V \to D, \gamma : D \to V \rangle$ is said to be an adjunction between the complete lattices $V = \langle V, \sqsubseteq_V, \bigsqcup_V, \bigsqcap_V, \bot_V, \top_V \rangle$ and $D = \langle D, \sqsubseteq_D, \bigsqcup_D, \bigsqcap_D, \bot_D, \top_D \rangle$ iff α and γ are total functions that satisfy

$$\alpha(v) \sqsubseteq_D d \iff v \sqsubseteq_V \gamma(d)$$

for all $v \in V$ and $d \in D$. ◀

In fact, adjunctions are Galois connections.

▶ **Theorem 3.5** (Adjunctions and Galois connections). *$\langle \alpha : V \to D, \gamma : D \to V \rangle$ is an adjunction iff it is a Galois connection.* ◀

Proof. First assume that $\langle \alpha : V \to D, \gamma : D \to V \rangle$ is an adjunction. It will be proven that it also is a Galois connection by showing that $\gamma \circ \alpha \sqsupseteq_V \lambda v.v$ and $\alpha \circ \gamma \sqsubseteq_D \lambda d.d$. For any $v \in V$, trivially $\alpha(v) \sqsubseteq_D \alpha(v)$. Using that $\alpha(v) \sqsubseteq_D d \Rightarrow v \sqsubseteq_V \gamma(d)$, it can be established that $v \sqsubseteq_V \gamma(\alpha(v))$. Similarly, for any $d \in D$, trivially $\gamma(d) \sqsubseteq_V \gamma(d)$. Using that $v \sqsubseteq_V \gamma(d) \Rightarrow \alpha(v) \sqsubseteq_D d$, it can be established that $\alpha(\gamma(d)) \sqsubseteq_D d$. Thus, $\langle \alpha : V \to D, \gamma : D \to V \rangle$ is a Galois connection.

Next assume that $\langle \alpha : V \to D, \gamma : D \to V \rangle$ is a Galois connection. It will be proven that it also is an adjunction by showing that $\alpha(v) \sqsubseteq_D d \Rightarrow v \sqsubseteq_V \gamma(d)$ and $v \sqsubseteq_V \gamma(d) \Rightarrow \alpha(v) \sqsubseteq_D d$. So, first assume that $\alpha(v) \sqsubseteq_D d$. Then, since γ is monotone, $\gamma(\alpha(v)) \sqsubseteq_V \gamma(d)$. Using that $\gamma \circ \alpha \sqsupseteq_V \lambda v.v$, it can be established that $v \sqsubseteq_V \gamma(\alpha(v)) \sqsubseteq_V \gamma(d)$ as required. For the second part of the proof, assume that $v \sqsubseteq_V \gamma(d)$. Then, since α is monotone, $\alpha(v) \sqsubseteq_D \alpha(\gamma(d))$. Using that $\alpha \circ \gamma \sqsubseteq_D \lambda d.d$, it can be established that $\alpha(v) \sqsubseteq_D \alpha(\gamma(d)) \sqsubseteq_D d$ as required. ◀

A Galois connection can be constructed, or its correctness can be verified, using the following theorems.

▶ **Theorem 3.6** (Galois connection – independent attribute method). *If* $\langle \alpha : V \to D, \gamma : D \to V \rangle$ *and* $\langle \alpha' : V' \to D', \gamma' : D' \to V' \rangle$ *are Galois connections, then so is* $\langle \alpha'' : (V \times V') \to (D \times D'), \gamma'' : (D \times D') \to (V \times V') \rangle$*, where*

$$\begin{cases} \alpha''((v, v')) = (\alpha(v), \alpha'(v')) \\ \gamma''((d, d')) = (\gamma(d), \gamma'(d')) \end{cases}$$

and $(v, v') \in V \times V'$ *and* $(d, d') \in D \times D'$. ◀

Proof. First calculate the following.

$$\begin{aligned} \alpha''((v, v')) \sqsubseteq (d, d') &\overset{def.\ \alpha''}{\iff} (\alpha(v), \alpha'(v')) \sqsubseteq (d, d') \\ &\overset{calc.}{\iff} \alpha(v) \sqsubseteq d \wedge \alpha'(v') \sqsubseteq d' \\ &\overset{th.\ 3.5}{\iff} v \sqsubseteq \gamma(d) \wedge v' \sqsubseteq \gamma'(d') \\ &\overset{calc.}{\iff} (v, v') \sqsubseteq (\gamma(d), \gamma'(d')) \\ &\overset{def.\ \gamma''}{\iff} (v, v') \sqsubseteq \gamma''((d, d')) \end{aligned}$$

Then, using Theorem 3.5, the result follows. ◀

▶ **Theorem 3.7** (Galois connection – lifted independent attribute method). *If* $\langle \alpha_1 : \mathcal{P}(V_1) \to D_1, \gamma_1 : D_1 \to \mathcal{P}(V_1) \rangle$ *and* $\langle \alpha_2 : \mathcal{P}(V_2) \to D_2, \gamma_2 : D_2 \to \mathcal{P}(V_2) \rangle$ *are Galois connections, then so is* $\langle \alpha : \mathcal{P}(V_1 \times V_2) \to (D_1 \times D_2), \gamma : (D_1 \times D_2) \to \mathcal{P}(V_1 \times V_2) \rangle$*, where*

$$\begin{cases} \alpha(V) = (\alpha_1(\{v_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V\}), \alpha_2(\{v_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V\})) \\ \gamma((d_1, d_2)) = \gamma_1(d_1) \times \gamma_2(d_2) \end{cases}$$

and $V \subseteq V_1 \times V_2$ *and* $(d_1, d_2) \in D_1 \times D_2$. ◀

Proof. First, calculate

$$\begin{aligned} \alpha(V) \sqsubseteq (d_1, d_2) &\overset{def.\ \alpha}{\iff} (\alpha_1(V_1'), \alpha_2(V_2')) \sqsubseteq (d_1, d_2) \\ &\overset{calc.}{\iff} \alpha_1(V_1') \sqsubseteq_1 d_1 \wedge \alpha_2(V_2') \sqsubseteq_2 d_2 \\ &\overset{th.\ 3.5}{\iff} V_1' \subseteq \gamma_1(d_1) \wedge V_2' \subseteq \gamma_2(d_2) \\ &\overset{calc.}{\iff} V_1' \times V_2' \subseteq \gamma_1(d_1) \times \gamma_2(d_2) \\ &\overset{def.\ \gamma}{\iff} V_1' \times V_2' \subseteq \gamma((d_1, d_2)) \\ &\overset{V \subseteq V_1' \times V_2'}{\iff} V \subseteq \gamma((d_1, d_2)) \end{aligned}$$

where $V_1' = \{v_1 \mid \exists v_2 \in V_2 : (v_1, v_2) \in V\}$ and $V_2' = \{v_2 \mid \exists v_1 \in V_1 : (v_1, v_2) \in V\}$. Then, using Theorem 3.5, the result follows. ◀

▶ **Theorem 3.8** (Galois connection – double lifting). *If* $\langle \alpha : V \to D, \gamma : D \to V \rangle$ *is a Galois connection, then so is* $\langle \alpha_{\mathcal{P}} : \mathcal{P}(V) \to \mathcal{P}(D), \gamma_{\mathcal{P}} : \mathcal{P}(D) \to \mathcal{P}(V) \rangle$*, where*

$$\begin{cases} \alpha_{\mathcal{P}}(V') = \{\alpha(v) \mid v \in V'\} \\ \gamma_{\mathcal{P}}(D') = \{\gamma(d) \mid d \in D'\} \end{cases}$$

and $V' \subseteq V$ *and* $D' \subseteq D$. ◀

Proof. First, note that $\gamma_{\mathcal{P}}$ is monotone since γ is. Then calculate the following.

$$\begin{array}{lll}
\alpha_{\mathcal{P}}(V') \sqsubseteq D' & \overset{def.\ \alpha_{\mathcal{P}}}{\iff} & \{\alpha(v) \mid v \in V'\} \sqsubseteq D' \\
& \overset{calc.}{\iff} & \forall v \in V' : \{\alpha(v)\} \sqsubseteq D' \\
& \overset{\gamma_{\mathcal{P}}\ mon.}{\iff} & \forall v \in V' : \gamma_{\mathcal{P}}(\{\alpha(v)\}) \sqsubseteq \gamma_{\mathcal{P}}(D') \\
& \overset{def.\ \gamma_{\mathcal{P}}}{\iff} & \forall v \in V' : \{\gamma(\alpha(v))\} \sqsubseteq \gamma_{\mathcal{P}}(D') \\
& \overset{\lambda v.v \sqsubseteq \gamma \circ \alpha}{\iff} & \forall v \in V' : \{v\} \sqsubseteq \gamma_{\mathcal{P}}(D') \\
& \overset{calc.}{\iff} & V' \sqsubseteq \gamma_{\mathcal{P}}(D')
\end{array}$$

Then, using Theorem 3.5, the result follows. ◀

▶ **Theorem 3.9** (Galois connection – total function space). *If $\langle \alpha : V \to D, \gamma : D \to V \rangle$ is a Galois connection, then so is $\langle \alpha' : (S \to V) \to (S \to D), \gamma' : (S \to D) \to (S \to V) \rangle$ for some set S, where:*

$$\begin{cases} \alpha'(f) = \alpha \circ f \\ \gamma'(g) = \gamma \circ g \end{cases}$$ ◀

Proof. First note that α' and γ' are monotone since α and γ are. Furthermore, since $\langle \alpha, \gamma \rangle$ is a Galois connection,

$$\begin{array}{l} \gamma'(\alpha'(f)) = \gamma \circ \alpha \circ f \sqsupseteq f \\ \alpha'(\gamma'(g)) = \alpha \circ \gamma \circ g \sqsubseteq g \end{array}$$

and, thus, the theorem holds. ◀

▶ **Theorem 3.10** (Galois connection – lifted total function space). *If $\langle \alpha : \mathcal{P}(V) \to D, \gamma : D \to \mathcal{P}(V) \rangle$ is a Galois connection, then so is $\langle \alpha_s : \mathcal{P}(S \to V) \to (S \to D), \gamma_s : (S \to D) \to \mathcal{P}(S \to V) \rangle$, for some set S, where*

$$\begin{cases} \alpha_s(V') = \lambda s \in S.\alpha(\{v'\ s \mid v' \in V'\}) \\ \gamma_s(d) = \{\lambda s \in S.v \mid v \in \gamma(d\ s)\} \end{cases}$$

and $V' \subseteq S \to V$ and $d \in S \to D$. ◀

Proof. First note that γ_s is monotone because γ is. Then calculate the following.

$$\begin{array}{lll}
\alpha_s(V') \sqsubseteq d & \overset{def.\ \alpha_s}{\iff} & \lambda s \in S.\alpha(\{v'\ s \mid v' \in V'\}) \sqsubseteq d \\
& \overset{\gamma_s\ mon.}{\iff} & \gamma_s(\lambda s \in S.\alpha(\{v'\ s \mid v' \in V'\})) \sqsubseteq \gamma_s(d) \\
& \overset{def.\ \gamma_s}{\iff} & \{\lambda s \in S.v \mid v \in \gamma((\lambda s' \in S.\alpha(\{v'\ s' \mid v' \in V'\}))\ s)\} \sqsubseteq \gamma_s(d) \\
& \overset{calc.}{\iff} & \{\lambda s \in S.v \mid v \in \gamma(\alpha(\{v'\ s \mid v' \in V'\}))\} \sqsubseteq \gamma_s(d) \\
& \overset{\lambda v.v \sqsubseteq \gamma \circ \alpha}{\iff} & \{\lambda s \in S.v \mid v \in \{v'\ s \mid v' \in V'\}\} \sqsubseteq \\
& & \qquad \{\lambda s \in S.v \mid v \in \gamma(\alpha(\{v'\ s \mid v' \in V'\}))\} \sqsubseteq \gamma_s(d) \\
& \overset{calc.}{\iff} & \{\lambda s \in S.v \mid v \in \{v'\ s \mid v' \in V'\}\} \sqsubseteq \gamma_s(d) \\
& \overset{calc.}{\iff} & \{\lambda s \in S.(v'\ s) \mid v' \in V'\} \sqsubseteq \gamma_s(d) \\
& \overset{calc.}{\iff} & \{v' \mid v' \in V'\} \sqsubseteq \gamma_s(d) \\
& \overset{calc.}{\iff} & V' \sqsubseteq \gamma_s(d)
\end{array}$$

Then, using Theorem 3.5, the result follows. ◀

▶ **Theorem 3.11** (Galois connection – indexing). *If $\langle \alpha : V \to D, \gamma : D \to V \rangle$ is a Galois connection, then so is $\langle \alpha' : (S \times V) \to (S \times D), \gamma' : (S \times D) \to (S \times V) \rangle$, for some partially ordered set $S \ni s$, with the partial order defined as $s \sqsubseteq s' \iff s = s'$, where*

$$\begin{cases} \alpha'((s,v)) = (s, \alpha(v)) \\ \gamma'((s',d)) = (s', \gamma(d)) \end{cases}$$

and $(s,v) \in S \times V$ and $(s',d) \in S \times D$. ◀

Proof. First calculate the following.

$$\begin{aligned} \alpha'((s,v)) \sqsubseteq (s',d) &\overset{def.\ \alpha'}{\iff} (s, \alpha(v)) \sqsubseteq (s',d) \\ &\overset{calc.}{\iff} s = s' \wedge \alpha(v) \sqsubseteq d \\ &\overset{th.\ 3.5}{\iff} s = s' \wedge v \sqsubseteq \gamma(d) \\ &\overset{calc.}{\iff} (s,v) \sqsubseteq (s', \gamma(d)) \\ &\overset{def.\ \gamma'}{\iff} (s,v) \sqsubseteq \gamma'((s',d)) \end{aligned}$$

Then, using Theorem 3.5, the result follows. ◀

3.2 Constructing a Galois Insertion

A Galois insertion $\langle \alpha, \gamma \rangle$ between two domains, D and $\tilde{D}$, can be constructed by following steps 1-5 below [5].

1. A domain D with a partial order $\sqsubseteq$, a least (bottom) element $\bot$, a greatest (top) element $\top$, a greatest lower bound $\bigsqcap$ and a least upper bound $\bigsqcup$, so that $\langle D, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top \rangle$ is a complete lattice must be given.
2. Define a domain $\tilde{D}$ and a monotone concretisation function $\gamma : \tilde{D} \to D$.
3. Define the partial order $\tilde{\sqsubseteq}$ for $\tilde{D}$.
4. The greatest lower bound $\tilde{\bigsqcap}$ and the least upper bound $\tilde{\bigsqcup}$ must exist for all subsets of $\tilde{D}$. Then, by definition, $\langle \tilde{D}, \tilde{\sqsubseteq}, \tilde{\bigsqcup}, \tilde{\bigsqcap}, \tilde{\bot}, \tilde{\top} \rangle$ is a complete lattice.
5. Define the abstraction function $\alpha : D \to \tilde{D}$, which must be monotone.

Assuming that the domains D and $\tilde{D}$ and the monotone concretisation function γ are defined, the partial ordering $\tilde{\sqsubseteq}$ can easily be defined as [5]:

▶ **Definition 3.12** (Partial order). $\tilde{\sqsubseteq}$ is a partial order for the domain $\tilde{D}$ iff $\forall \tilde{d}_1, \tilde{d}_2 \in \tilde{D} : (\tilde{d}_1 \tilde{\sqsubseteq} \tilde{d}_2 \iff \gamma(\tilde{d}_1) \sqsubseteq \gamma(\tilde{d}_2))$. ◀

Based on this definition of the partial order, the greatest lower bound and least upper bound can be defined [5].

▶ **Definition 3.13** (Greatest lower bound). The element $\tilde{d} \in \tilde{D}$ is a lower bound of $\tilde{D}' \subseteq \tilde{D}$ iff $\forall \tilde{d}' \in \tilde{D}' : \tilde{d} \tilde{\sqsubseteq} \tilde{d}'$. The element $\tilde{d} \in \tilde{D}$ is the greatest lower bound of $\tilde{D}' \subseteq \tilde{D}$ ($\tilde{d} = \tilde{\bigsqcap} \tilde{D}'$) iff $\tilde{d}$ is a lower bound of $\tilde{D}'$ and for all other lower bounds $\tilde{d}'$ of $\tilde{D}'$ we have $\tilde{d}' \tilde{\sqsubseteq} \tilde{d}$. ◀

▶ **Definition 3.14** (Least upper bound). The element $\tilde{d} \in \tilde{D}$ is an upper bound of $\tilde{D}' \subseteq \tilde{D}$ iff $\forall \tilde{d}' \in \tilde{D}' : \tilde{d}' \tilde{\sqsubseteq} \tilde{d}$. The element $\tilde{d} \in \tilde{D}$ is the least upper bound of $\tilde{D}' \subseteq \tilde{D}$ ($\tilde{d} = \tilde{\bigsqcup} \tilde{D}'$) iff $\tilde{d}$ is an upper bound of $\tilde{D}'$ and for all other upper bounds $\tilde{d}'$ of $\tilde{D}'$ we have $\tilde{d} \tilde{\sqsubseteq} \tilde{d}'$. ◀

The abstraction function α can be defined based on the definition of the greatest lower bound operator [5].

▶ **Definition 3.15** (Abstraction function, α)**.** Given two domains D and $\tilde{D}$ and a monotone concretisation function $\gamma : \tilde{D} \to D$, the abstraction function $\alpha : D \to \tilde{D}$ is defined by:

$$\alpha(d) = \tilde{\bigsqcap}\{\tilde{d} \mid d \sqsubseteq \gamma(\tilde{d})\}$$

where $d \in D$ and $\tilde{d} \in \tilde{D}$. ◀

Alternatively, assuming that we have defined two domains and a monotone abstraction function, the concretisation function γ can be defined based on the least upper bound operator.

▶ **Definition 3.16** (Alternative definition – Concretisation function, γ)**.** Given two domains D and $\tilde{D}$ and a monotone abstraction function $\alpha : D \to \tilde{D}$, the concretisation function $\gamma : \tilde{D} \to D$ is defined by:

$$\gamma(\tilde{d}) = \bigsqcup\{d \mid \alpha(d) \mathrel{\tilde{\sqsubseteq}} \tilde{d}\}$$

where $d \in D$ and $\tilde{d} \in \tilde{D}$. ◀

3.3 The Interval Domain

One example of an abstract domain for values is the interval domain [4, 5, 10]. The definition of an interval is given in Definition 3.17.

▶ **Definition 3.17** (Interval)**.** An interval is defined as $[n_1, n_2]$, where $n_1, n_2 \in \mathbf{Val} = \mathbb{Z} \cup \{-\infty, \infty\}$ are the lower and upper bounds of the interval, respectively, and $n_1 \leq n_2$. Formally, the set of all intervals is defined as $\mathbf{Intv} = \{\bot_{int}\} \cup \{[n_1, n_2] \mid n_1 \leq n_2 \wedge n_1, n_2 \in \mathbf{Val}\}$, where $\bot_{int}$ denotes an invalid interval and $\top_{int} = [-\infty, \infty]$. ◀

A Galois insertion will now be created between $\mathcal{P}(\mathbf{Val})$ and $\mathbf{Intv}$, using the steps of Section 3.2. The concretisation function $\gamma_{int} : \mathbf{Intv} \to \mathcal{P}(\mathbf{Val})$ is defined as:

▶ **Definition 3.18** (Concretisation of intervals)**.**

$$\gamma_{int}(i) = \begin{cases} \emptyset & \textbf{if } i = \bot_{int} \\ \{n \in \mathbf{Val} \mid n_1 \leq n \leq n_2\} & \textbf{if } i = [n_1, n_2] \end{cases}$$ ◀

The partial order relation for intervals, $\sqsubseteq_{int}$, is defined as (using Definition 3.12):

▶ **Definition 3.19** (Partial order for intervals)**.**

$$\begin{cases} i \sqsubseteq_{int} \top_{int} \\ \bot_{int} \sqsubseteq_{int} i \\ [n_1, n_2] \sqsubseteq_{int} [n_1', n_2'] \iff n_1' \leq n_1 \wedge n_2 \leq n_2' \end{cases}$$ ◀

The greatest lower bound operator for intervals $\bigsqcap_{int}$ is defined as (using Definition 3.13):

▶ **Definition 3.20** (Greatest lower bound for intervals)**.**

$$\begin{cases} i \sqcap_{int} \top_{int} = \top_{int} \sqcap_{int} i = i \\ i \sqcap_{int} \bot_{int} = \bot_{int} \sqcap_{int} i = \bot_{int} \\ [n_1, n_2] \sqcap_{int} [n_1', n_2'] = \\ \qquad \begin{cases} [\max(\{n_1, n_1'\}), \min(\{n_2, n_2'\})] & \textbf{if } \max(\{n_1, n_1'\}) \leq \min(\{n_2, n_2'\}) \\ \bot_{int} & \textbf{otherwise} \end{cases} \end{cases}$$ ◀

The least upper bound operator for intervals $\bigsqcup_{int}$ is defined as (using Definition 3.14):

▶ **Definition 3.21** (Least upper bound for intervals)**.**

$$\begin{cases} i \sqcup_{int} \top_{int} = \top_{int} \sqcup_{int} i = \top_{int} \\ i \sqcup_{int} \bot_{int} = \bot_{int} \sqcup_{int} i = i \\ [n_1, n_2] \sqcup_{int} [n_1', n_2'] = [\min(\{n_1, n_1'\}), \max(\{n_2, n_2'\})] \end{cases}$$ ◀

The abstraction function $\alpha_{int} : \mathcal{P}(\mathbf{Val}) \to \mathbf{Intv}$ is defined as (using Definition 3.15):

▶ **Definition 3.22** (Abstraction to interval)**.**

$$\alpha_{int}(V) = \begin{cases} \bot_{int} & \textbf{if } V = \emptyset \\ [\min(V), \max(V)] & \textbf{otherwise} \end{cases}$$ ◀

To show that $\langle \alpha_{int}, \gamma_{int} \rangle$ is a Galois insertion, it would suffice to show that γ_{int} is monotone, since the steps of Section 3.2 have been used. However, for clarity, the entire proof will be provided.

▶ **Lemma 3.23** (Monotonicity of γ_{int})**.** *The function $\gamma_{int} : \mathbf{Intv} \to \mathcal{P}(\mathbf{Val})$ is monotone.* ◀

Proof. It should be shown that $\forall i, i' \in \mathbf{Intv} : (i \sqsubseteq_{int} i' \Rightarrow \gamma_{int}(i) \subseteq \gamma_{int}(i'))$.

For the case that $i = \bot_{int}$, the proof is trivial. Assume that $i = [n_1, n_2] \in \mathbf{Intv}$ and $i' = [n_1', n_2'] \in \mathbf{Intv}$, such that $i \sqsubseteq_{int} i'$. Further assume that $n \in \gamma_{int}(i)$. Then it must be the case that $n_1 \leq n \leq n_2$ (Definition 3.18). Since $i \sqsubseteq_{int} i'$, it must be the case that $n_1' \leq n_1 \leq n \leq n_2 \leq n_2'$ (Definition 3.19). But, then it must be that $n \in \gamma_{int}(i')$ (Definition 3.18), and thus, $\gamma_{int}(i) \subseteq \gamma_{int}(i')$. ◀

▶ **Lemma 3.24** (Monotonicity of α_{int})**.** *The function $\alpha_{int} : \mathcal{P}(\mathbf{Val}) \to \mathbf{Intv}$ is monotone.* ◀

Proof. It should be shown that $\forall V, V' \in \mathcal{P}(\mathbf{Val}) : (V \subseteq V' \Rightarrow \alpha_{int}(V) \sqsubseteq_{int} \alpha_{int}(V'))$.

For the case that $V = \emptyset$, the proof is trivial. Assume that $V, V' \in \mathcal{P}(\mathbf{Val})$, such that $V \subseteq V'$. Further assume that $\alpha_{int}(V) = [n_1, n_2]$ and $\alpha_{int}(V') = [n_1', n_2']$. Since $V \subseteq V'$, it must be that $\forall v \in V : \{v\} \subseteq V'$, and hence, $\{n_1, n_2\} \subseteq V'$. But then, it must be that $\min(V') = n_1' \leq n_1 = \min(V)$ and $\max(V) = n_2 \leq n_2' = \max(V')$, and thus, $[n_1, n_2] \sqsubseteq_{int} [n_1', n_2']$ (Definition 3.19), which means that $\alpha_{int}(V) \sqsubseteq_{int} \alpha_{int}(V')$. ◀

▶ **Theorem 3.25** (Galois insertion)**.** $\langle \alpha_{int} : \mathcal{P}(\mathbf{Val}) \to \mathbf{Intv}, \gamma_{int} : \mathbf{Intv} \to \mathcal{P}(\mathbf{Val}) \rangle$ *is a Galois insertion.* ◀

Proof. The proof amounts to showing that Definition 3.2 is fulfilled by $\langle \alpha_{int}, \gamma_{int} \rangle$. Note that $\mathcal{P}(\mathbf{Val})$ and $\mathbf{Intv}$ are complete lattices [10].

According to Lemmas 3.23 and 3.24, γ_{int} and α_{int} are monotone. To show that $\alpha_{int}(\gamma_{int}(i)) = i$, assume that $i \in \mathbf{Intv}$.

- If $i = \bot_{int}$, then $\gamma_{int}(i) = \emptyset$. Thus, $\alpha_{int}(\gamma_{int}(i)) = \alpha_{int}(\emptyset) = \bot_{int} = i$.
- If $i = [n_1, n_2]$, then $\gamma_{int}(i) = \{n \in \mathbf{Val} \mid n_1 \leq n \leq n_2\}$. Thus, $\alpha_{int}(\gamma_{int}(i)) = \alpha_{int}(\{n \in \mathbf{Val} \mid n_1 \leq n \leq n_2\}) = [n_1, n_2] = i$.

To show that $\gamma_{int}(\alpha_{int}(V)) \supseteq V$, assume that $V \in \mathcal{P}(\mathbf{Val})$.

- If $V = \emptyset$, then $\alpha_{int}(V) = \bot_{int}$. Thus, $\gamma_{int}(\alpha_{int}(V)) = \gamma_{int}(\bot_{int}) = \emptyset \supseteq \emptyset = V$.
- If $V \neq \emptyset$, then $\alpha_{int}(V) = [\min(V), \max(V)]$. Thus, $\gamma_{int}(\alpha_{int}(V)) = \gamma_{int}([\min(V), \max(V)]) = \{n \in \mathbf{Val} \mid \min(V) \leq n \leq \max(V)\} \supseteq V$. ◀

$$
\begin{array}{ll}
P ::= T \mid P \parallel T & s ::= [\texttt{halt}]^l \mid [\texttt{skip}]^l \mid [r \texttt{ := } a]^l \mid [\texttt{if } b \texttt{ goto } l']^l \mid s_1 \texttt{ ; } s_2 \mid \\
 & \quad [\texttt{load } r \texttt{ from } x]^l \mid [\texttt{store } r \texttt{ to } x]^l \mid [\texttt{lock } lck]^l \mid [\texttt{unlock } lck]^l \\
T ::= (N, s) & a ::= n \mid r \mid a_1 \texttt{ + } a_2 \mid a_1 \texttt{ - } a_2 \mid a_1 \texttt{ * } a_2 \mid a_1 \texttt{ / } a_2 \\
 & b ::= \texttt{true} \mid \texttt{false} \mid \texttt{!}b \mid b_1 \texttt{ \&\& } b_2 \mid a_1 \texttt{ == } a_2 \mid a_1 \texttt{ <= } a_2
\end{array}
$$

Figure 1 The parallel programming language.

$\text{STM}(T, pc)$	$\langle pc', \mathbb{r}', \mathbb{x}', \mathbb{l}' \rangle$	Condition
$[\texttt{halt}]^{pc}$	$\langle pc, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{skip}]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	–
$[r \texttt{ := } a]^{pc}$	$\langle pc+1, \mathbb{r}[r \mapsto \mathcal{A}[\![a]\!]\mathbb{r}], \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{load } r \texttt{ from } x]^{pc}$	$\langle pc+1, \mathcal{R}(r, \mathbb{r}, x, \mathbb{x}), \mathbb{x}, \mathbb{l} \rangle$	–
$[\texttt{store } r \texttt{ to } x]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}[x \mapsto (\mathbb{x}\ x)[T \mapsto \{(\mathbb{r}\ r, t)\}]], \mathbb{l} \rangle$	–
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\neg\mathcal{B}[\![b]\!]\mathbb{r}$
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle l, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\mathcal{B}[\![b]\!]\mathbb{r}$
$[\texttt{lock } lck]^{pc}$	$\langle pc, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$	$\text{OWN}(\mathbb{l}\ lck) \neq T$
$[\texttt{lock } lck]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l}[lck \mapsto (locked, T)] \rangle$	$\text{OWN}(\mathbb{l}\ lck) = T$
$[\texttt{unlock } lck]^{pc}$	$\langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l}[lck \mapsto (unlocked, \bot_{thrd})] \rangle$	–

where $\mathcal{R}(r, \mathbb{r}, x, \mathbb{x}) = \mathbb{r}[r \mapsto v]$ and for some $t' \in \textbf{Time}$, $\{(v, t')\} = \bigcup_{T' \in \textbf{Thrd}}((\mathbb{x}\ x)\ T')$

Figure 2 Semantics of concrete axiom transitions: $\langle T, pc, \mathbb{r}, \mathbb{x}, \mathbb{l}, t \rangle \underset{ax}{\longrightarrow} \langle pc', \mathbb{r}', \mathbb{x}', \mathbb{l}' \rangle$

4 Timing Analysis

In this section, an algorithm for timing analysis of programs containing *dependent* parallel threads will be defined. It is assumed that the underlying architecture consists of both thread-private and global memory, referred to as registers, $r \in \textbf{Reg}$, and variables, $x \in \textbf{Var}$, respectively, and that arithmetical operations etc. can be performed using values of registers. It will also be assumed that shared resources that can be acquired in a mutually exclusive manner by the threads are provided, and that the operations provided by the instruction set (statements) may have variable execution times. (C.f., multicore CPU:s, where you have local and global memory, a shared memory bus and mutual exclusion operations.)

4.1 A Parallel Programming Language

The analysis will be based on the parallel programming language defined in Figure 1, which is a set of operations using the discussed architectural features. $P \in \textbf{Prg}$ denotes a program, which simply is a number of threads, denoted by $T \in \textbf{Thrd}$. A thread is a pair of a statement, $s \in \textbf{Stm}$, and a unique identifier, $N \in \textbf{ThrdID}$. This makes every thread unique and distinguishable from other threads, even if several threads contain the same statement. To increase the readability of the semantics, it will be assumed that the axiom-statements (all statements except the sequentially composed statement, $s_1 \texttt{ ; } s_2$) of each thread are uniquely labelled with consecutive labels, $l \in \textbf{Lbl}$, and stored in an array-like fashion in ascending order of their labels. $a \in \textbf{Aexp}$ and $b \in \textbf{Bexp}$ denote an arithmetic and a boolean expression, respectively, $n \in \textbf{Val}$ is an integer value, and $lck \in \textbf{Lck}$ denotes a lock. Locks can be acquired in a mutually exclusive manner using `lock` and released using `unlock`. Values can be transferred between variables and registers using `load` and `store`. Conditional branching is performed using `if`, a register is assigned a value using `:=`, a no-operation is performed using `skip`, and `halt` stops the execution of the issuing thread. The arithmetical, boolean and relational operators are self-explanatory and will not be discussed further.

The semantics of the language is formally defined in Figures 2 (individual axiom state-

$$\textsc{stm}((N,s),pc) = \begin{cases} s & \textbf{if } s = [\texttt{skip}]^{pc} \vee \\ & \quad s = [r \texttt{ := } a]^{pc} \vee \\ & \quad s = [\texttt{if } b \texttt{ goto } l']^{pc} \vee \\ & \quad s = [\texttt{load } r \texttt{ from } x]^{pc} \vee \\ & \quad s = [\texttt{store } r \texttt{ to } x]^{pc} \vee \\ & \quad s = [\texttt{lock } lck]^{pc} \vee \\ & \quad s = [\texttt{unlock } lck]^{pc} \vee \\ & \quad s = [\texttt{halt}]^{pc} \\ \textsc{stm}((N,s'),pc) & \textbf{if } s = s';s'' \wedge pc \in \textsc{labels}(s') \\ \textsc{stm}((N,s''),pc) & \textbf{if } s = s';s'' \wedge pc \in \textsc{labels}(s'') \end{cases}$$

(a) Definition of STM.

$$\textsc{labels}(s) = \begin{cases} \{l\} & \textbf{if } s = [\texttt{skip}]^{l} \vee \\ & \quad s = [r \texttt{ := } a]^{l} \vee \\ & \quad s = [\texttt{if } b \texttt{ goto } l']^{l} \vee \\ & \quad s = [\texttt{load } r \texttt{ from } x]^{l} \vee \\ & \quad s = [\texttt{store } r \texttt{ to } x]^{l} \vee \\ & \quad s = [\texttt{lock } lck]^{l} \vee \\ & \quad s = [\texttt{unlock } lck]^{l} \vee \\ & \quad s = [\texttt{halt}]^{l} \\ \textsc{labels}(s') \cup \textsc{labels}(s'') & \textbf{if } s = s';s'' \end{cases}$$

(b) Definition of LABELS.

Figure 3 Definition of STM and LABELS.

$$\frac{\forall T \in \mathbf{Thrd}_{exe} : \langle T, pc_T, \mathbb{r}_T, \mathbb{x}, \mathbb{l}'', t_T^{a\prime}\rangle \xrightarrow[ax]{} \langle pc'_T, \mathbb{r}'_T, \mathbb{x}'_T, \mathbb{l}'_T\rangle}{\langle \{(T, pc_T, \mathbb{r}_T, t_T^r, t_T^a) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t\rangle \xrightarrow[prg]{} \langle \{(T, pc'_T, \mathbb{r}'_T, t_T^{r\prime}, t_T^{a\prime}) \mid T \in \mathbf{Thrd}\}, \mathbb{x}', \mathbb{l}', t'\rangle}$$

where

$$t_T^{r\prime} = \begin{cases} \textsc{finTime}(\langle \{(T, pc_T, \mathbb{r}_T, t_T^r, t_T^a) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t\rangle, T) & \textbf{if } t = t_T^a \\ t_T^r & \textbf{otherwise} \end{cases}$$

$$t' = \min(\{t_T^a + t_T^{r\prime} \mid T \in \mathbf{Thrd}\})$$

$$t_T^{a\prime} = \begin{cases} t_T^a + t_T^{r\prime} & \textbf{if } t' = t_T^a + t_T^{r\prime} \\ t_T^a & \textbf{otherwise} \end{cases}$$

$$\mathbf{Thrd}_{exe} = \{T \in \mathbf{Thrd} \mid t' = t_T^{a\prime}\}$$

$$(\mathbb{x}'\ x)\ T = \begin{cases} \begin{cases} (\mathbb{x}'_T\ x)\ T & \text{for some } T \in \mathbf{Thrd}_{exe} : \exists r \in \mathbf{Reg}_T : \textsc{stm}(T, pc_T) = [\texttt{store } r \texttt{ to } x]^{pc_T} \\ \emptyset & \text{for } T' \in \mathbf{Thrd} \setminus \{T\}, \textbf{ if such a } T \text{ exists} \end{cases} \\ (\mathbb{x}\ x)\ T & \textbf{otherwise} \end{cases}$$

$$\mathbb{l}''\ lck = \begin{cases} (\mathit{unlocked}, T) & \text{for some } T \in \mathbf{Thrd}_{exe} : \textsc{stm}(T, pc_T) = [\texttt{lock } lck]^{pc_T}, \textbf{ if such} \\ & T \text{ exists, } \textsc{stt}(\mathbb{l}\ lck) = \mathit{unlocked} \text{ and } \textsc{own}(\mathbb{l}\ lck) = \bot_{thrd} \\ \mathbb{l}\ lck & \textbf{otherwise} \end{cases}$$

$$\mathbb{l}'\ lck = \begin{cases} \mathbb{l}'_T\ lck & \text{for some } T \in \mathbf{Thrd}_{exe} : (\textsc{stm}(T, pc_T) = [\texttt{unlock } lck]^{pc_T} \vee \\ & (\textsc{own}(\mathbb{l}''\ lck) = T \wedge \textsc{stm}(T, pc_T) = [\texttt{lock } lck]^{pc_T})), \textbf{ if such } T \text{ exists} \\ \mathbb{l}\ lck & \textbf{otherwise} \end{cases}$$

Figure 4 Semantics of concrete program transitions: $\langle \mathbf{Ts}, \mathbb{x}, \mathbb{l}, t\rangle \xrightarrow[prg]{} \langle \mathbf{Ts}', \mathbb{x}', \mathbb{l}', t'\rangle$

$$\mathcal{A}[\![n]\!]\mathbb{r} = n \qquad \mathcal{A}[\![a_1 \texttt{ + } a_2]\!]\mathbb{r} = \mathcal{A}[\![a_1]\!]\mathbb{r} + \mathcal{A}[\![a_2]\!]\mathbb{r} \qquad \mathcal{A}[\![a_1 \texttt{ * } a_2]\!]\mathbb{r} = \mathcal{A}[\![a_1]\!]\mathbb{r} \cdot \mathcal{A}[\![a_2]\!]\mathbb{r}$$
$$\mathcal{A}[\![r]\!]\mathbb{r} = \mathbb{r}\ r \qquad \mathcal{A}[\![a_1 \texttt{ - } a_2]\!]\mathbb{r} = \mathcal{A}[\![a_1]\!]\mathbb{r} - \mathcal{A}[\![a_2]\!]\mathbb{r} \qquad \mathcal{A}[\![a_1 \texttt{ / } a_2]\!]\mathbb{r} = \left\lfloor \frac{\mathcal{A}[\![a_1]\!]\mathbb{r}}{\mathcal{A}[\![a_2]\!]\mathbb{r}} \right\rfloor$$

Figure 5 Semantics of concrete evaluation of arithmetic expressions.

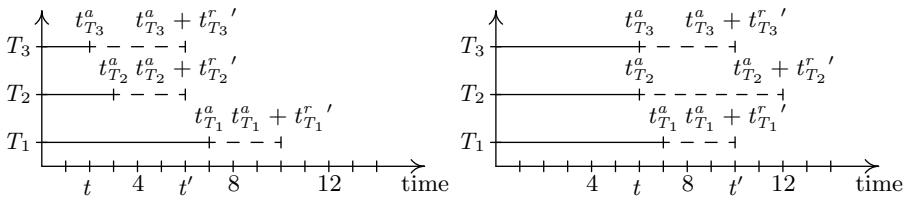
$$\mathcal{B}[\![\texttt{true}]\!]\mathbb{r} \iff \texttt{true} \qquad \mathcal{B}[\![\texttt{!}\, b]\!]\mathbb{r} \iff \neg\mathcal{B}[\![b]\!]\mathbb{r} \qquad \mathcal{B}[\![a_1 \texttt{ == } a_2]\!]\mathbb{r} \iff \mathcal{A}[\![a_1]\!]\mathbb{r} = \mathcal{A}[\![a_2]\!]\mathbb{r}$$
$$\mathcal{B}[\![\texttt{false}]\!]\mathbb{r} \iff \texttt{false} \qquad \mathcal{B}[\![b_1 \texttt{ \&\& } b_2]\!]\mathbb{r} \iff \mathcal{B}[\![b_1]\!]\mathbb{r} \wedge \mathcal{B}[\![b_2]\!]\mathbb{r} \qquad \mathcal{B}[\![a_1 \texttt{ <= } a_2]\!]\mathbb{r} \iff \mathcal{A}[\![a_1]\!]\mathbb{r} \leq \mathcal{A}[\![a_2]\!]\mathbb{r}$$

Figure 6 Semantics of concrete evaluation of boolean expressions.

ments) and 4 (system of threads). $\mathbb{x} \in \mathbf{Var} \to \mathbf{Thrd} \to \mathcal{P}(\mathbf{Val} \times \mathbf{Time})$, $\mathbb{l} \in \mathbf{Lck} \to (\mathbf{Lck_{stt}} \times \mathbf{Thrd} \cup \{\bot_{thrd}\})$, where $\mathbf{Lck_{stt}} = \{unlocked, locked\}$, and $t \in \mathbf{Time}$ are the states for variables and locks, and the current time. For each thread, T, in the program, there is also $pc_T \in \mathbf{Lbl}_T$, $\mathbb{r}_T \in \mathbf{Reg}_T \to \mathbf{Val}$, $t^r_T \in \mathbf{Time}$ and $t^a_T \in \mathbf{Time}$, which are the states of the program counter and registers of T, the relative execution time of T's active statement, $\textsc{stm}(T, pc_T)$ (note that STM is defined in Figure 3a and LABELS is defined in Figure 3b), and the accumulated execution time for T, respectively. The tuple collecting all these states will be referred to as a configuration, c, i.e., $c = \langle\{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t\rangle$. Note that states are updated on transitions, i.e., when pc is updated.

The state for locks keeps track of the state and owner of each lock. The owner is $\bot_{thrd}$ if no thread currently has the lock acquired. The state for registers of thread T simply keeps track of the current value of each register within T. The state for variables is not as intuitive. To be precise, the abstraction of the state for variables will need to save write history, i.e., what abstract writes (a pair of value and time) have been performed by each thread on each variable (see Section 4.2). Therefore, to derive a Galois connection (and hence implicitly get a safe approximation [10]), the concrete state for variables has to be defined accordingly. In the concrete semantics, only one single write is saved for each variable, though. This write is non-deterministically chosen from one of the threads, if any, writing the variable at any given point in time (see Figure 4). $\mathcal{R}$ is defined to return the value of the saved write (see Figure 2).

$\mathcal{A} : \mathbf{Aexp} \to (\mathbf{Reg} \to \mathbf{Val}) \to \mathbf{Val}$ and $\mathcal{B} : \mathbf{Bexp} \to (\mathbf{Reg} \to \mathbf{Val}) \to \mathbf{Bool}$ evaluate arithmetic and boolean expressions, respectively, given a particular register state. The semantics of these functions can be found in Figures 5 and 6, respectively. FINTIME is assumed to be provided by a timing-model of the underlying hardware. It should return a relative execution time for the statement of thread T, i.e., $\textsc{stm}(T, pc_T)$, based on the current system state. The set of threads to execute, $\mathbf{Thrd}_{exe}$, is determined based on t', $t^{r\prime}$ and t^a. It simply consists of the threads that will update their pc:s at the nearest point in time, t'. An illustration of how ${t^r_T}'$, t^a_T, t, t' and $\mathbf{Thrd}_{exe}$ are related is given in Figure 7. For c_1 in Figure 7a, $t' = t^a_{T_2} + {t^r_{T_2}}' = t^a_{T_3} + {t^r_{T_3}}' = 6$. Thus, $T_2, T_3 \in \mathbf{Thrd}_{exe}$ and will hence update their t^a. For c_2 in Figure 7b, ${t^r_{T_2}}'$ and ${t^r_{T_3}}'$ are updated using FINTIME to again determine



(a) c_1: $\mathbf{Thrd}_{exe} = \{T_2, T_3\}$ **(b)** c_2: $\mathbf{Thrd}_{exe} = \{T_1, T_3\}$

Figure 7 Illustration of how $\mathbf{Thrd}_{exe}$ is determined ($c_1 \xrightarrow[prg]{} c_2$).

$\mathbf{Thrd}_{exe}$.

The behaviour of locks needs to be explained. Assume that some threads in $\mathbf{Thrd}_{exe}$ execute a `lock`-statement on some lock, *lck*, and that *lck* is *unlocked* in the given configuration. In the resulting configuration, $\textsc{stt}(\mathbb{l}'\ lck) = locked$ and the owner will be one of the threads that tried to acquire *lck*. The chosen thread is given by $\textsc{own}(\mathbb{l}''\ lck)$; note that $\mathbb{l}''$ is only used to control the behaviour of the rules for `lock` in Figure 2. This thread will have incremented its *pc* and thus moved on to executing its next statement. All other threads that tried to acquire *lck* will again try to acquire *lck* since their *pc*:s are not changed. Note that the latter would also be the case for *all* threads in $\mathbf{Thrd}_{exe}$ that try to acquire an already locked lock that is not owned by themselves. Also note that a thread who owns a lock is allowed to repeatedly acquire this lock any number of times.

4.2 Abstract Interpretation

First, it must be decided what parts of the system state to interpret in an abstract way. To allow for the hardware timing-model to be approximated as well, **Time** will be abstracted using the interval domain, i.e., $\mathbf{\tilde{Time}} = \mathbf{Intv}$. This approach is also taken by Chattopadhyay et al. [1] to approximate the execution time of pipeline stages in order to deal with timing anomalies in multicore platforms. **Val** will also be abstracted using intervals, i.e., $\mathbf{\tilde{Val}} = \mathbf{Intv}$, to allow for an efficient handling of data flow. Since **Thrd**, **Lbl**, **Var**, **Reg**, **Lck**, **Aexp** and **Bexp** are defined by the software, it does not make any sense to abstract them for the defined analysis (see Section 4.3). And, since $\mathbf{Lck_{stt}}$ is comparable to **Bool**, an abstraction of it would not be very beneficial. The states implicitly affected by the abstractions of **Time** and **Val** are $\mathbb{r}$, $\mathbb{x}$, t^r, t^a, t, and thus c. The abstraction of these will be referred to as $\tilde{\mathbb{r}}$, $\tilde{\mathbb{x}}$, $\tilde{t}^r$, $\tilde{t}^a$, $\tilde{t}$ and $\tilde{c}$, respectively.

Since values are abstracted using the interval domain, the operators of the language must be extended to act on intervals. This is done in Figure 8. Note that ∞/∞, $0/0$, $0 * \infty$ and $\infty - \infty$ need not be defined.

Using Theorems 3.10 and 3.25, it is easy to see that there is indeed a Galois connection, $\langle \alpha_{reg}, \gamma_{reg} \rangle$, between the domains $\mathcal{P}(\mathbf{Reg} \to \mathbf{Val})$ and $\mathbf{Reg} \to \mathbf{\tilde{Val}}$. The concretisation and abstraction functions, partial order and least upper and greatest lower bounds are given by Definitions 4.1, 4.2, 4.3, 4.4 and 4.5. The bottom element $\tilde{\bot}_{reg} \in \mathbf{Reg} \to \mathbf{\tilde{Val}}$ corresponds to an abstract mapping for which some register maps to $\bot_{int}$. This does not mean that there exist several bottom elements. Rather, if some register maps to $\bot_{int}$ in $\tilde{\mathbb{r}}$, then $\tilde{\mathbb{r}}$ is pulled down to become the bottom element, i.e., $\tilde{\mathbb{r}} = \tilde{\bot}_{reg}$. The top element corresponds to an abstract mapping for which all registers map to $\top_{int}$.

▶ **Definition 4.1** (Concretisation of an abstract register store).

$$\gamma_{reg}(\tilde{\mathbb{r}}) = \begin{cases} \emptyset & \textbf{if } \tilde{\mathbb{r}} = \tilde{\bot}_{reg} \\ \{\lambda r \in \mathbf{Reg}.v \mid v \in \gamma_{int}(\tilde{\mathbb{r}}\ r)\} & \textbf{otherwise} \end{cases}$$ ◀

▶ **Definition 4.2** (Partial order for abstract register stores).

$$\begin{cases} \tilde{\bot}_{reg} \mathrel{\tilde{\sqsubseteq}}_{reg} \tilde{\mathbb{r}} \\ \tilde{\mathbb{r}} \mathrel{\tilde{\sqsubseteq}}_{reg} \tilde{\mathbb{r}}' \iff \forall r \in \mathbf{Reg} : \tilde{\mathbb{r}}\ r \sqsubseteq_{int} \tilde{\mathbb{r}}'\ r \end{cases}$$ ◀

▶ **Definition 4.3** (Greatest lower bound of abstract register stores).

$$\begin{cases} \tilde{\bot}_{reg} \mathrel{\tilde{\sqcap}}_{reg} \tilde{\mathbb{r}} = \tilde{\mathbb{r}} \mathrel{\tilde{\sqcap}}_{reg} \tilde{\bot}_{reg} = \tilde{\bot}_{reg} \\ (\tilde{\mathbb{r}} \mathrel{\tilde{\sqcap}}_{reg} \tilde{\mathbb{r}}')\ r = (\tilde{\mathbb{r}}\ r) \sqcap_{int} (\tilde{\mathbb{r}}'\ r) \end{cases}$$ ◀

$$
\begin{aligned}
[n_1,n_2] +_{int} [n_1',n_2'] = \begin{cases}
[n_1+n_1', n_2+n_2'] & \textbf{if } -\infty < n_1, n_1', n_2, n_2' < \infty \\
[n_1+n_1', \infty] & \textbf{if } (n_2 = \infty \vee n_2' = \infty) \wedge \\
 & \quad \neg((n_1 = -\infty \wedge n_1' = \infty) \vee \\
 & \quad\quad (n_1 = \infty \wedge n_1' = -\infty)) \\
[-\infty, n_2+n_2'] & \textbf{if } (n_1 = -\infty \vee n_1' = -\infty) \wedge \\
 & \quad \neg((n_2 = -\infty \wedge n_2' = \infty) \vee \\
 & \quad\quad (n_2 = \infty \wedge n_2' = -\infty)) \\
[-\infty, \infty] & \textbf{otherwise}
\end{cases} \\
[n_1,n_2] -_{int} [n_1',n_2'] = \begin{cases}
[n_1-n_2', n_2-n_1'] & \textbf{if } -\infty < n_1, n_1', n_2, n_2' < \infty \\
[n_1-n_2', \infty] & \textbf{if } (n_2 = \infty \vee n_1' = -\infty) \wedge \\
 & \quad \neg((n_1 = \infty \wedge n_2' = \infty) \vee \\
 & \quad\quad (n_1 = -\infty \wedge n_2' = -\infty)) \\
[-\infty, n_2-n_1'] & \textbf{if } (n_1 = -\infty \vee n_2' = \infty) \wedge \\
 & \quad \neg((n_2 = \infty \wedge n_1' = \infty) \vee \\
 & \quad\quad (n_2 = -\infty \wedge n_1' = -\infty)) \\
[-\infty, \infty] & \textbf{otherwise}
\end{cases} \\
[n_1,n_2] *_{int} [n_1',n_2'] = \begin{cases}
[\min(V), \max(V)] & \textbf{if } (n_2 < 0 \wedge n_1' > 0) \vee \\
 & \quad (n_2 < 0 \wedge n_2' < 0) \vee \\
 & \quad (n_1 > 0 \wedge n_1' > 0) \vee \\
 & \quad (n_1 > 0 \wedge n_2' < 0) \vee \\
 & \quad (-\infty < n_1, n_1', n_2, n_2' < \infty) \\
 & \quad \textbf{where } V = \{n_1 * n_1', n_1 * n_2', n_2 * n_1', n_2 * n_2'\} \\
[-\infty, \infty] & \textbf{otherwise}
\end{cases} \\
[n_1,n_2] /_{int} [n_1',n_2'] = \begin{cases}
[\lfloor \min(V) \rfloor, \lceil \max(V) \rceil] & \textbf{if } (-\infty < n_1' \wedge n_2' < 0) \vee \\
 & \quad (0 < n_1' \wedge n_2' < \infty) \\
 & \quad \textbf{where } V = \{n_1/n_1', n_1/n_2', n_2/n_1', n_2/n_2'\} \\
[-\infty, -n_1] & \quad \textbf{if } n_1' \leq 0 \leq n_2' \wedge n_2 < 0 \\
[-n_2, \infty] & \quad \textbf{if } n_1' \leq 0 \leq n_2' \wedge 0 < n_1 \\
[-\infty, \infty] & \quad \textbf{otherwise}
\end{cases}
\end{aligned}
$$

Figure 8 Language operators defined for interval arguments.

$$\tilde{\mathcal{A}}[\![a]\!]\,\tilde{\bot}_{reg} = \tilde{\bot}_{val} \qquad \tilde{\mathcal{A}}[\![a_1 + a_2]\!]\tilde{\mathtt{r}} = \tilde{\mathcal{A}}[\![a_1]\!]\tilde{\mathtt{r}} +_{int} \tilde{\mathcal{A}}[\![a_2]\!]\tilde{\mathtt{r}}$$
$$\tilde{\mathcal{A}}[\![n]\!]\tilde{\mathtt{r}} = n \qquad \tilde{\mathcal{A}}[\![a_1 - a_2]\!]\tilde{\mathtt{r}} = \tilde{\mathcal{A}}[\![a_1]\!]\tilde{\mathtt{r}} -_{int} \tilde{\mathcal{A}}[\![a_2]\!]\tilde{\mathtt{r}}$$
$$\tilde{\mathcal{A}}[\![r]\!]\tilde{\mathtt{r}} = \tilde{\mathtt{r}}\ r \qquad \tilde{\mathcal{A}}[\![a_1 * a_2]\!]\tilde{\mathtt{r}} = \tilde{\mathcal{A}}[\![a_1]\!]\tilde{\mathtt{r}} *_{int} \tilde{\mathcal{A}}[\![a_2]\!]\tilde{\mathtt{r}}$$
$$\tilde{\mathcal{A}}[\![a_1 \,/\, a_2]\!]\tilde{\mathtt{r}} = \tilde{\mathcal{A}}[\![a_1]\!]\tilde{\mathtt{r}} /_{int} \tilde{\mathcal{A}}[\![a_2]\!]\tilde{\mathtt{r}}$$

Figure 9 The abstract function evaluating arithmetic expressions.

▶ **Definition 4.4** (Least upper bound of abstract register stores)**.**

$$\begin{cases} \tilde{\bot}_{reg}\,\tilde{\sqcup}_{reg}\,\tilde{\mathtt{r}} = \tilde{\mathtt{r}}\,\tilde{\sqcup}_{reg}\,\tilde{\bot}_{reg} = \tilde{\mathtt{r}} \\ (\tilde{\mathtt{r}}\,\tilde{\sqcup}_{reg}\,\tilde{\mathtt{r}}')\ r = (\tilde{\mathtt{r}}\ r) \sqcup_{int} (\tilde{\mathtt{r}}'\ r) \end{cases}$$ ◀

▶ **Definition 4.5** (Abstraction of a set of register stores)**.**

$$\alpha_{reg}(\mathbb{R}) = \begin{cases} \tilde{\bot}_{reg} & \textbf{if } \mathbb{R} = \emptyset \\ \lambda r \in \textbf{Reg}.\alpha_{int}(\{\mathtt{r}\ r \mid \mathtt{r} \in \mathbb{R}\}) & \textbf{otherwise} \end{cases}$$ ◀

The function evaluating arithmetic expressions, $\mathcal{A}$, must be abstracted since values and register stores are abstracted. The abstraction will be $\tilde{\mathcal{A}} : \textbf{Aexp} \to (\textbf{Reg} \to \textbf{Vãl}) \to \textbf{Vãl}$ and can be derived using Definition 3.3 to induce $\mathcal{A}$. To do this, $\mathcal{A}$ must first be lifted to sets of concrete register mappings:

$$\mathcal{A}_{\mathcal{P}}[\![a]\!]\mathbb{R} = \{\mathcal{A}[\![a]\!]\mathtt{r} \mid \mathtt{r} \in \mathbb{R}\}$$

The abstract evaluation function can then be derived as:

$$\tilde{\mathcal{A}}[\![a]\!] = \alpha_{val} \circ \mathcal{A}_{\mathcal{P}}[\![a]\!] \circ \gamma_{reg}$$

The details of this function can be found in Figure 9.

The function $\tilde{\mathcal{BR}}$, defined in Definition 4.6, will be used in the abstract axiom transition rules (see Figure 11). This function is safely induced from $\mathcal{B}$, using Definition 3.3, so that the concretisation of $\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathtt{r}}$ always contains (at least) the concrete stores, derived from $\tilde{\mathtt{r}}$, in which b evaluates to `true`.

▶ **Definition 4.6** (Boolean restriction)**.**

$$\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathtt{r}} = \alpha_{reg}(\{\mathtt{r} \in \gamma_{reg}(\tilde{\mathtt{r}}) \mid \mathcal{B}[\![b]\!]\mathtt{r}\})$$ ◀

Using Theorems 3.7, 3.8, 3.10 and 3.25, it is easy to see that there is indeed a Galois connection, $\langle \alpha_{var}, \gamma_{var} \rangle$, between the domains $\mathcal{P}(\textbf{Var} \to \textbf{Thrd} \to \mathcal{P}(\textbf{Val} \times \textbf{Time}))$ and $\textbf{Var} \to \textbf{Thrd} \to \mathcal{P}(\textbf{Vãl} \times \textbf{Tim̃e})$. The concretisation and abstraction functions, partial order and least upper and greatest lower bounds are given by Definitions 4.9, 4.15, 4.16, 4.17 and 4.10. The bottom element $\tilde{\bot}_{var} \in \textbf{Var} \to \textbf{Thrd} \to \mathcal{P}(\textbf{Vãl} \times \textbf{Tim̃e})$ corresponds to a mapping for which the set of abstract writes for some variable and thread contains $(\tilde{\bot}_{val}, \tilde{t})$, $(\tilde{v}, \tilde{\bot}_t)$ or $(\tilde{\bot}_{val}, \tilde{\bot}_t)$. This does not mean that there exist several bottom elements. Rather, if the set of writes for some variable and thread in $\tilde{\mathtt{x}}$ contains one of these elements, then $\tilde{\mathtt{x}}$ is pulled down to become the bottom element, i.e., $\tilde{\mathtt{x}} = \tilde{\bot}_{var}$. The top element corresponds to a mapping for which the set of writes for all variables and threads contain the element $(\tilde{\top}_{val}, \tilde{\top}_t)$. $\tilde{\mathtt{x}} \in \textbf{Var} \to \textbf{Thrd} \to \mathcal{P}(\textbf{Vãl} \times \textbf{Tim̃e})$ can save any number (i.e., history) of abstract writes, $\tilde{w} \in \textbf{Vãl} \times \textbf{Tim̃e}$, for each thread that occur on some variable. This is done to increase the precision in the analysis, since then, sequence (within each thread) and timing information (between threads) can be used to get a tight value when reading a variable.

For convenience in expressing, and readability of, the upcoming algorithms, some relations for abstract writes will be defined. The partial order for writes, $\tilde{\sqsubseteq}_w$, follows naturally (c.f., Definition 3.12) from the partial orders for values, $\tilde{\sqsubseteq}_{val}$, and time, $\tilde{\sqsubseteq}_t$.

▶ **Definition 4.7** (Partial order of writes)**.**

$$\begin{cases} \tilde{\bot}_w \;\tilde{\sqsubseteq}_w\; \tilde{w} \\ (\tilde{v}_1, \tilde{t}_1) \;\tilde{\sqsubseteq}_w\; (\tilde{v}_2, \tilde{t}_2) \iff \tilde{v}_1 \;\tilde{\sqsubseteq}_{val}\; \tilde{v}_2 \wedge \tilde{t}_1 \;\tilde{\sqsubseteq}_t\; \tilde{t}_2 \end{cases}$$ ◀

The precedence relation, $\tilde{<}_t$, on abstract times can be useful to determine whether two writes are performed at disjunct times.

▶ **Definition 4.8** (Time precedence)**.**

$$\begin{cases} \tilde{\bot}_t \;\tilde{<}_t\; \tilde{t} \\ \tilde{t}_1 \;\tilde{<}_t\; \tilde{t}_2 \iff \max(\gamma_t(\tilde{t}_1)) < \min(\gamma_t(\tilde{t}_2)) \end{cases}$$ ◀

The concretisation of abstract variable stores is decribed by γ_{var} which is presented in Definition 4.9.

▶ **Definition 4.9** (Concretization of an abstract variable store)**.**

$$\gamma_{var}(\tilde{\mathbb{x}}) = \{\lambda x \in \mathbf{Var}.f \mid f \in \{\lambda T \in \mathbf{Thrd}.W \mid W \in \{\gamma_{val}(\tilde{v}) \times \gamma_t(\tilde{t}) \mid (\tilde{v}, \tilde{t}) \in ((\tilde{\mathbb{x}}\ x)\ T)\}\}\}$$ ◀

The abstraction of concrete variable stores is decribed by α_{var} which is presented in Definition 4.10.

▶ **Definition 4.10** (Abstraction of a concrete variable store)**.**

$$\alpha_{var}(\mathbb{X}) = \lambda x \in \mathbf{Var}.\lambda T \in \mathbf{Thrd}.\{(\alpha_{val}(\{v \mid \exists t \in \mathbf{Time} : (v,t) \in W\}), \alpha_t(\{t \mid \exists v \in \mathbf{Val} : (v,t) \in W\})) \mid W \in \{(\mathbb{x}\ x)\ T \mid \mathbb{x} \in \mathbb{X}\}\}$$ ◀

Note that the definitions of $\tilde{\sqsubseteq}_{var}$, $\tilde{\bigsqcap}_{var}$ and $\tilde{\bigsqcup}_{var}$ could follow naturally from the definition of the domain (i.e., they could be defined based on $\subseteq$, $\bigcap$ and $\bigcup$, respectively). However, this will not be the case. This is due to the fact that the history in the two stores might have different traces (sequence information) and can thus not simply be, e.g., joined. Instead, the operations to be used should be defined based on Definition 4.11 to ensure that all threads see safe values (see Definition 4.12) at all times.

▶ **Definition 4.11** (Safe write history)**.** An abstract variable store, $\tilde{\mathbb{x}}$, is safe at time $\tilde{t}$ if it contains, at least, all the abstract values (not writes) that might be seen by some thread at time $\tilde{t}$; i.e., it covers at least all the possible concrete writes that might have occurred at time $t \in \gamma_t(\tilde{t})$.

Thus, to be safe, $\tilde{\mathbb{x}}$ must, for each variable and each thread, contain at least:

1. all writes, $(\tilde{v}, \tilde{t}')$, such that $\tilde{t} \,\tilde{\sqcap}_t\, \tilde{t}' \neq \tilde{\bot}_t$, and
2. the latest (most recent) write, $(\tilde{v}, \tilde{t}')$, such that $\tilde{t} \,\tilde{\sqcap}_t\, \tilde{t}' = \tilde{\bot}_t$. The most recent write(s), $(\tilde{v}, \tilde{t}')$, is defined such that $\min(\gamma_t(\tilde{t})) > \max(\gamma_t(\tilde{t}')) > \max(\gamma_t(\tilde{t}''))$, for all other writes, $(\tilde{v}', \tilde{t}'')$, for which $\tilde{t} \,\tilde{\sqcap}_t\, \tilde{t}'' = \tilde{\bot}_t$. ◀

▶ **Definition 4.12** (Safe value of x as seen by thread T)**.** Assuming that $\tilde{\mathbb{x}}$ contains safe write history for all threads on variable x, according to Definition 4.11, a safe value of x, as seen by thread T, at time $\tilde{t}$ is the least upper bound, $\tilde{\bigsqcup}_{val}$, of the values of at least the following writes.

1. All writes, $\tilde{w}_{T'} = (\tilde{v}_{T'}, \tilde{t}_{T'})$, for all threads $T' \in \mathbf{Thrd} \setminus \{T\}$ such that $\tilde{t}_{T'} \,\tilde{\sqcap}_t\, \tilde{t} \neq \tilde{\bot}_t$.

Algorithm 4.1 Write to Variable

```
1: function WRITE(T, x̃, x, w̃)
2:     if x̃ = ⊥̃_var ∨ w̃ = ⊥̃_w then
3:         x̃′ ← ⊥̃_var
4:     else
5:         (x̃′ x′) T′ ← { ((x̃ x) T) ∪ {w̃}   if x′ = x ∧ T = T′
                        { ((x̃ x′) T′)        otherwise
6:     end if
7:     return x̃′
8: end function
```

2. The most recent write $\tilde{w}'_{T'} = (\tilde{v}'_{T'}, \tilde{t}'_{T'})$, for each thread $T' \in \mathbf{Thrd}$, among the remaining writes (i.e., $\tilde{w}'_{T'} \neq \tilde{w}_{T'}$), if $\tilde{t}'_{T'} \tilde{\sqcap}_t \tilde{t}_{mrw} \neq \tilde{\bot}_t$, where $\tilde{t}_{mrw} \tilde{<}_t \tilde{t}$ is the time of the most recent write of any write, not belonging to 1. ◀

WRITE$(T, \tilde{\mathbb{x}}, x, \tilde{w})$, as defined in Algorithm 4.1, safely (Lemma 4.13) adds the write, $\tilde{w}$, to the set of write-history for thread T, i.e., to $((\tilde{\mathbb{x}}\ x)\ T)$.

▶ **Lemma 4.13** (Safety of WRITE)**.** *Assuming that $\tilde{\mathbb{x}}$ contains safe write history for thread T before the write is performed, which occurs at time $\tilde{t}$ (c.f., Definition 4.11), then so will* WRITE$(T, \tilde{\mathbb{x}}, x, (\tilde{v}, \tilde{t}))$. ◀

Proof. Since WRITE$(T, \tilde{\mathbb{x}}, x, (\tilde{v}, \tilde{t}))$ simply adds the write $(\tilde{v}, \tilde{t})$ to the history of thread T's writes on variable x in the store $\tilde{\mathbb{x}}$, and $\tilde{\mathbb{x}}$ is assumed to contain safe write history, WRITE$(T, \tilde{\mathbb{x}}, x, (\tilde{v}, \tilde{t}))$ is trivially safe. ◀

Algorithm 4.2 Read from Variable

```
1: function READ(x̃, x, T, t̃)
2:     for all T′ ∈ Thrd \ {T} do
3:         ((x̃ x) T′) ← {(ṽ, t̃′) ∈ ((x̃ x) T′) | t̃ ⋠_t t̃′}
4:     end for
5:     ((x̃ x) T) ← {(ṽ, t̃′) ∈ ((x̃ x) T) | min(γ_t(t̃)) ≥ min(γ_t(t̃′))}
6:     W ← ∅
7:     for all T′ ∈ Thrd \ {T} do
8:         W_T′ ← {(ṽ′, t̃′) ∈ ((x̃ x) T′) | t̃′ ⊓̃_t t̃ ≠ ⊥̃_t}
9:         ((x̃′ x) T′) ← ((x̃ x) T′) \ W_T′
10:        W ← W_T′ ∪ W
11:    end for
12:    ((x̃′ x) T) ← ((x̃ x) T)
13:    t̃_mrw ← MOSTRECENTWRITETIME(x̃′, x)
14:    if t̃_mrw ≠ ⊥̃_t then
15:        for all T′ ∈ Thrd do
16:            t̃_mrw_T′ ← MOSTRECENTWRITETIMETHREAD((x̃′ x) T)
17:            W ← W ∪ {(ṽ′, t̃′) ∈ ((x̃′ x) T′) | t̃′ = t̃_mrw_T′ ∧ t̃′ ⊓̃_t t̃_mrw ≠ ⊥̃_t}
18:        end for
19:    end if
20:    ṽ ← ⨆̃_val {ṽ′ | ∃t̃′ ∈ Tim̃e : (ṽ′, t̃′) ∈ W}
21:    return ṽ
22: end function
```

Using the sequence and timing information provided by Definition 4.11, READ$(\tilde{\mathbb{x}}, x, T, \tilde{t})$, as defined in Algorithm 4.2, only takes the writes that might be valid at $\tilde{t}$ (the point in

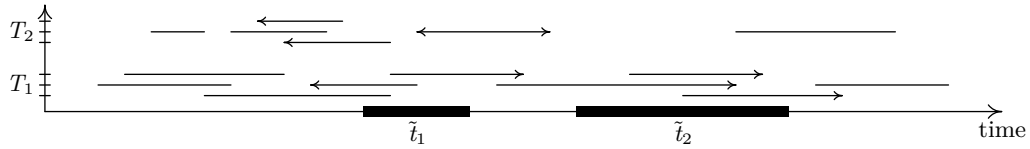


Figure 10 The time-stamps of the writes considered by READ($\tilde{\mathrm{x}}, x, T_1, \tilde{t}_1$) and READ($\tilde{\mathrm{x}}, x, T_2, \tilde{t}_2$).

Algorithm 4.3 Time of Most Recent Write

```
1: function MOSTRECENTWRITETIME(x̃, x)
2:     t̃_mrw ← ⊥̃_t
3:     for all T ∈ Thrd do
4:         t̃_mrw_T ← MOSTRECENTWRITETIMETHREAD((x̃ x) T)
5:         if t̃_mrw = ⊥̃_t then
6:             t̃_mrw ← t̃_mrw_T
7:         else if t̃_mrw_T ≠ ⊥̃_t then
8:             if max(γ_t(t̃_mrw_T)) > max(γ_t(t̃_mrw)) then
9:                 t̃_mrw ← t̃_mrw_T
10:            else if max(γ_t(t̃_mrw_T)) = max(γ_t(t̃_mrw)) then
11:                t̃_mrw ← t̃_mrw_T ⊓̃_t t̃_mrw
12:            end if
13:        end if
14:    end for
15:    return t̃_mrw
16: end function
```

Algorithm 4.4 Time of Most Recent Write in Thread

```
1: function MOSTRECENTWRITETIMETHREAD(set)
2:     if set = ∅ then
3:         return ⊥̃_t
4:     end if
5:     t_max ← max(⋃{γ_t(t̃) | ∃ṽ ∈ Vãl : (ṽ, t̃) ∈ set})
6:     t_min ← min(⋂{γ_t(t̃) | ∃ṽ ∈ Vãl : (ṽ, t̃) ∈ set ∧ max(γ_t(t̃)) = t_max})
7:     return α_t({t_min, t_max})
8: end function
```

time when T issues the READ) into consideration for its returned value $\tilde{v} \in \mathbf{V\tilde{a}l}$. These writes, $\tilde{w} = (\tilde{v}', \tilde{t}')$, come from two categories. The first category covers the writes on x for threads $T' \in \mathbf{Thrd} \setminus \{T\}$ whose "time-stamps" overlap in time with $\tilde{t}$, i.e., $\tilde{t} \tilde{\sqcap}_t \tilde{t}' \neq \tilde{\bot}_t$. The second category covers the most recent write on x for all threads (including T) such that its time-stamp overlaps with the overall most recent write of any write, not belonging to the first category. Note that any write for thread T with a time-stamp that begins after the beginning of $\tilde{t}$ is discarded. So is any write for $T' \in \mathbf{Thrd} \setminus \{T\}$ such that its time-stamp completely succeeds $\tilde{t}$. This is because such writes can simply not have occurred at the time of the READ (and will thus usually not be included in $\tilde{\mathbb{x}}$ at all). An illustration of the time-stamps of the writes in T_1 and T_2 that must be considered by $\text{READ}(\tilde{\mathbb{x}}, x, T_1, \tilde{t}_1)$ (lines with arrow heads pointing left) and $\text{READ}(\tilde{\mathbb{x}}, x, T_2, \tilde{t}_2)$ (lines with arrow heads pointing right) is given in Figure 10. The returned value, $\tilde{v}$, is the least upper bound of the values of the considered writes. Note that MOSTRECENTWRITETIME and MOSTRECENTWRITETIMETHREAD are defined in Algorithms 4.3 and 4.4, respectively, based on the fact that time progresses between successive writes within each thread.

▶ **Lemma 4.14** (Safety of READ). *Assuming that $\tilde{\mathbb{x}}$ contains safe write history at $\tilde{t}$ (see Definition 4.11),* $\text{READ}(\tilde{\mathbb{x}}, x, T, \tilde{t})$ *returns a safe value (see Definition 4.12).* ◀

Proof. The proof amounts to showing that all writes described by Definition 4.12 is included in the resulting value. This is trivial since READ is derived using Definition 4.12. ◀

The partial order for abstract variable stores, $\tilde{\sqsubseteq}_{var}$, is defined based on PARTIALORDERVARIABLES in Algorithm 4.5, taking the safety of write history (Definition 4.11) into account. Note that EARLIESTWRITETHREAD is defined in Algorithm 4.6. The idea is that the history for each thread and variable should be the same in both stores for the relation to evaluate to true. However, the histories are allowed to differ somewhat. The greater store could also contain newer writes than those in the history of the lesser store. It could also be the case that the newest write in the greater store is, at least, the least upper bound of the newest writes in the lesser store that are not part of both histories.

▶ **Definition 4.15** (Partial order of abstract variable stores).

$$\begin{cases} \tilde{\bot}_{var} \tilde{\sqsubseteq}_{var} \tilde{\mathbb{x}} \\ \tilde{\mathbb{x}} \tilde{\sqsubseteq}_{var} \tilde{\mathbb{x}}' \iff \text{PARTIALORDERVARIABLES}(\tilde{\mathbb{x}}, \tilde{\mathbb{x}}') \end{cases}$$ ◀

Based on the partial order relation, the greatest lower bound and least upper bound operators, $\tilde{\bigsqcap}_{var}$ and $\tilde{\bigsqcup}_{var}$, respectively, can be defined. MEETVARIABLES is defined in Algorithm 4.7 and JOINVARIABLES is defined in Algorithm 4.8.

▶ **Definition 4.16** (Greatest lower bound of abstract variable stores).

$$\begin{cases} \tilde{\bot}_{var} \tilde{\sqcap}_{var} \tilde{\mathbb{x}} = \tilde{\mathbb{x}} \tilde{\sqcap}_{var} \tilde{\bot}_{var} = \tilde{\bot}_{var} \\ \tilde{\mathbb{x}} \tilde{\sqcap}_{var} \tilde{\mathbb{x}}' = \text{MEETVARIABLES}(\tilde{\mathbb{x}}, \tilde{\mathbb{x}}') \end{cases}$$ ◀

▶ **Definition 4.17** (Least upper bound of abstract variable stores).

$$\begin{cases} \tilde{\bot}_{var} \tilde{\sqcup}_{var} \tilde{\mathbb{x}} = \tilde{\mathbb{x}} \tilde{\sqcup}_{var} \tilde{\bot}_{var} = \tilde{\bot}_{var} \\ \tilde{\mathbb{x}} \tilde{\sqcup}_{var} \tilde{\mathbb{x}}' = \text{JOINVARIABLES}(\tilde{\mathbb{x}}, \tilde{\mathbb{x}}') \end{cases}$$ ◀

Note that neither $\tilde{\sqsubseteq}_{var}$, $\tilde{\bigsqcap}_{var}$ nor $\tilde{\bigsqcup}_{var}$ is used by the analysis but are just presented for completeness in the definition of the lattice. However, if merging of configurations [6] is introduced to lower the complexity of the analysis, at least $\tilde{\bigsqcup}_{var}$ will be needed.

Algorithm 4.5 Partial Order of Abstract Variable Stores

```
1: function partialOrderVariables(x̃, x̃′)
2:     for all x ∈ Var do
3:         for all T ∈ Thrd do
4:             ws ← ((x̃ x) T)
5:             ws′ ← ((x̃′ x) T)
6:             while ws ≠ ∅ ∧ ws′ ≠ ∅ do
7:                 w̃ ← earliestWriteThread(ws)
8:                 w̃′ ← earliestWriteThread(ws′)
9:                 ws ← ws \ {w̃}
10:                ws′ ← ws′ \ {w̃′}
11:                if w̃ ≠ w̃′ then
12:                    if ws′ = ∅ then
13:                        for all w̃″ ∈ ws ∪ {w̃} do
14:                            if w̃″ ⋤_w w̃′ then
15:                                return false
16:                            end if
17:                        end for
18:                    else
19:                        return false
20:                    end if
21:                end if
22:            end while
23:            if ws ≠ ∅ then
24:                return false
25:            end if
26:        end for
27:    end for
28:    return true
29: end function
```

Algorithm 4.6 Earliest Write for a Thread

```
1: function earliestWriteThread(set)
2:     if set = ∅ then
3:         return ⊥̃_w
4:     end if
5:     t̃_min ← α_t({∞})
6:     for all (ṽ, t̃) ∈ set do
7:         if min(γ_t(t̃)) < min(γ_t(t̃_min)) then
8:             t̃_min ← t̃
9:         else if min(γ_t(t̃)) = min(γ_t(t̃_min)) then
10:            t̃_min ← t̃ ⊓̃_t t̃_min
11:        end if
12:    end for
13:    W ← {(ṽ, t̃) | (ṽ, t̃) ∈ set ∧ t̃ = t̃_min}
14:    ṽ_min ← α_val({∞})
15:    for all (ṽ, t̃) ∈ W do
16:        if min(γ_val(ṽ)) < min(γ_val(ṽ_min)) then
17:            ṽ_min ← ṽ
18:        else if min(γ_val(ṽ)) = min(γ_val(ṽ_min)) then
19:            ṽ_min ← ṽ ⊓̃_val ṽ_min
20:        end if
21:    end for
22:    return (ṽ_min, t̃_min)
23: end function
```

Algorithm 4.7 Meeting Two Abstract Variable Stores

```
1: function meetVariables(x̃, x̃′)
2:     for all x ∈ Var do
3:         for all T ∈ Thrd do
4:             common ← ∅
5:             while ((x̃ x) T) ≠ ∅ ∧ ((x̃′ x) T) ≠ ∅ do
6:                 w̃ ← earliestWriteThread((x̃ x) T)
7:                 w̃′ ← earliestWriteThread((x̃′ x) T)
8:                 if w̃ = w̃′ then
9:                     common ← common ∪ {w̃}
10:                    ((x̃ x) T) ← ((x̃ x) T) \ {w̃}
11:                    ((x̃′ x) T) ← ((x̃′ x) T) \ {w̃′}
12:                else
13:                    ((x̃ x) T) ← ∅
14:                    ((x̃′ x) T) ← ∅
15:                end if
16:            end while
17:            ((x̃″ x) T) ← common
18:        end for
19:    end for
20:    return x̃″
21: end function
```

Algorithm 4.8 Joining Two Abstract Variable Stores

```
1:  function joinVariables(x̃, x̃′)
2:      for all x ∈ Var do
3:          for all T ∈ Thrd do
4:              common ← ∅
5:              merged ← ⊥̃_w
6:              while ((x̃ x) T) ≠ ∅ ∨ ((x̃′ x) T) ≠ ∅ do
7:                  w̃ ← earliestWriteThread((x̃ x) T)
8:                  w̃′ ← earliestWriteThread((x̃′ x) T)
9:                  if w̃ = w̃′ then
10:                     common ← common ∪ {w̃}
11:                     ((x̃ x) T) ← ((x̃ x) T) \ {w̃}
12:                     ((x̃′ x) T) ← ((x̃′ x) T) \ {w̃′}
13:                 else if ((x̃ x) T) = ∅ then
14:                     common ← common ∪ ((x̃′ x) T)
15:                     ((x̃′ x) T) ← ∅
16:                 else if ((x̃′ x) T) = ∅ then
17:                     common ← common ∪ ((x̃ x) T)
18:                     ((x̃ x) T) ← ∅
19:                 else
20:                     merged ← (⨆̃_w((x̃ x) T)) ⊔̃_w (⨆̃_w((x̃′ x) T))
21:                     ((x̃ x) T) ← ∅
22:                     ((x̃′ x) T) ← ∅
23:                 end if
24:             end while
25:             if merged = ⊥̃_w then
26:                 ((x̃″ x) T) ← common
27:             else
28:                 ((x̃″ x) T) ← common ∪ {merged}
29:             end if
30:         end for
31:     end for
32:     return x̃″
33: end function
```

The concrete domain for configurations is $\mathcal{P}(\mathbf{Conf})$. It is natural to define the partial ordering as $\subseteq$, the top element as $\mathbf{Conf}$, the bottom element as $\emptyset$, and the greatest lower bound and the least upper bound operators as $\cap$ and $\cup$, respectively. Then, by definition, $\langle \mathcal{P}(\mathbf{Conf}), \sqsubseteq_{conf}{=}\subseteq, \sqcup_{conf} = \cup, \sqcap_{conf} = \cap, \bot_{conf} = \emptyset, \top_{conf} = \mathbf{Conf} \rangle$ is a complete lattice.

The abstract domain for configurations is defined to be $\mathcal{P}(\mathbf{Co\tilde{n}f})$, since parts of the concrete configuration will not be abstracted; which means that any given set of concrete configurations cannot be translated into one single abstract configuration. It is thus natural to use $\subseteq$ as the partial ordering for this domain as well. By definition, $\langle \mathcal{P}(\mathbf{Co\tilde{n}f}), \tilde{\sqsubseteq}_{pconf}{=}\subseteq, \tilde{\sqcup}_{pconf} = \cup, \tilde{\sqcap}_{pconf} = \cap, \tilde{\bot}_{pconf} = \emptyset, \tilde{\top}_{pconf} = \mathbf{Co\tilde{n}f} \rangle$ is a complete lattice.

First a connection between the domains $\mathbf{Co\tilde{n}f}$ and $\mathcal{P}(\mathbf{Conf})$ will be introduced. Note that

$$\begin{aligned}\mathbf{Co\tilde{n}f} = \mathcal{P}(\mathbf{Thrd} \times \mathbf{Lbl} \times (\mathbf{Reg} \to \mathbf{V\tilde{a}l}) \times \mathbf{Ti\tilde{m}e} \times \mathbf{Ti\tilde{m}e}) \times (\mathbf{Var} \to \mathbf{Thrd} \to \\ \mathcal{P}(\mathbf{V\tilde{a}l} \times \mathbf{Ti\tilde{m}e})) \times (\mathbf{Lck} \to (\mathbf{Lck_{stt}} \times \mathbf{Thrd} \cup \{\bot_{thrd}\})) \times \mathbf{Ti\tilde{m}e}\end{aligned}$$

and thus

$$\tilde{c} ::= \langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}\}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}, \tilde{t} \rangle$$

The concretisation function for one single abstract configuration, $\gamma_{conf} : \mathbf{Co\tilde{n}f} \to \mathcal{P}(\mathbf{Conf})$, is defined as:

▶ **Definition 4.18** (Concretisation of one abstract configuration)**.**

$$\begin{cases} \gamma_{conf}(\tilde{\top}_{conf}) = \mathbf{Conf} \\ \gamma_{conf}(\tilde{\bot}_{conf}) = \emptyset \\ \gamma_{conf}(\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}, \tilde{t} \rangle) = \\ \quad \{\langle \{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid \\ \qquad T \in \mathbf{Thrd}_{\tilde{c}} \wedge \mathbb{r}_T \in \gamma_{reg}(\tilde{\mathbb{r}}_T) \wedge t^r_T \in \gamma_t(\tilde{t}^r_T) \wedge t^a_T \in \gamma_t(\tilde{t}^a_T)\}, \mathbb{x}, \mathbb{l}, t \rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbb{x} \in \gamma_{var}(\tilde{\mathbb{x}}) \wedge t \in \gamma_t(\tilde{t})\} \end{cases}$$ ◀

The partial ordering $\tilde{\sqsubseteq}_{conf}$ of two abstract configurations follows naturally using Definition 3.12.

▶ **Definition 4.19** (Partial ordering of two abstract configurations)**.**

$$\begin{cases} \tilde{\bot}_{conf} \;\tilde{\sqsubseteq}_{conf}\; \tilde{c} \\ \tilde{c} \;\tilde{\sqsubseteq}_{conf}\; \tilde{\top}_{conf} \\ \langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}, \tilde{t} \rangle \;\tilde{\sqsubseteq}_{conf} \\ \qquad\qquad \langle \{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^r{}'_T, \tilde{t}^a{}'_T) \mid T \in \mathbf{Thrd}_{\tilde{c}'}\}, \tilde{\mathbb{x}}', \tilde{\mathbb{l}}', \tilde{t}' \rangle \\ \qquad\qquad\qquad\qquad \iff \\ \qquad \tilde{\mathbb{x}} \;\tilde{\sqsubseteq}_{var}\; \tilde{\mathbb{x}}' \wedge \tilde{t} \;\tilde{\sqsubseteq}_t\; \tilde{t}' \wedge \tilde{\mathbb{l}} = \tilde{\mathbb{l}}' \wedge \mathbf{Thrd}_{\tilde{c}} = \mathbf{Thrd}_{\tilde{c}'} \wedge \\ \forall T \in \mathbf{Thrd}_{\tilde{c}} : (pc_T = pc'_T \wedge \tilde{\mathbb{r}}_T \;\tilde{\sqsubseteq}_{reg}\; \tilde{\mathbb{r}}'_T \wedge \tilde{t}^r_T \;\tilde{\sqsubseteq}_t\; \tilde{t}^r{}'_T \wedge \tilde{t}^a_T \;\tilde{\sqsubseteq}_t\; \tilde{t}^a{}'_T) \end{cases}$$ ◀

▶ **Lemma 4.20** (Monotonicity of γ_{conf})**.** *The function $\gamma_{conf} : \mathbf{Co\tilde{n}f} \to \mathcal{P}(\mathbf{Conf})$ is monotone with respect to $\tilde{\sqsubseteq}_{conf}$. I.e., if $\tilde{c}, \tilde{c}' \in \mathbf{Co\tilde{n}f}$ and $\tilde{c} \;\tilde{\sqsubseteq}_{conf}\; \tilde{c}'$, then $\gamma_{conf}(\tilde{c}) \subseteq \gamma_{conf}(\tilde{c}')$.* ◀

Proof. Assume that $\tilde{c}, \tilde{c}' \in \mathbf{Co\tilde{n}f}$ such that $\tilde{c} \;\tilde{\sqsubseteq}_{conf}\; \tilde{c}'$. If $\tilde{c} = \tilde{\bot}_{conf}$ or $\tilde{c}' = \tilde{\top}_{conf}$, the lemma holds trivially. Otherwise, assume that $\tilde{c} = \langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}, \tilde{t} \rangle$, $\tilde{c}' = \langle \{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^r{}'_T, \tilde{t}^a{}'_T) \mid T \in \mathbf{Thrd}_{\tilde{c}'}\}, \tilde{\mathbb{x}}', \tilde{\mathbb{l}}', \tilde{t}' \rangle$ and that $c \in \gamma_{conf}(\tilde{c})$. Since $\tilde{c} \;\tilde{\sqsubseteq}_{conf}\; \tilde{c}'$, it must be that:

$$\begin{gathered}\mathbf{Thrd}_{\tilde{c}} = \mathbf{Thrd}_{\tilde{c}'} \wedge \tilde{\mathbb{l}} = \tilde{\mathbb{l}}' \wedge \tilde{\mathbb{x}} \;\tilde{\sqsubseteq}_{var}\; \tilde{\mathbb{x}}' \wedge \tilde{t} \;\tilde{\sqsubseteq}_t\; \tilde{t}' \wedge \\ \forall T \in \mathbf{Thrd}_{\tilde{c}} : (pc_T = pc'_T \wedge \tilde{\mathbb{r}}_T \;\tilde{\sqsubseteq}_{reg}\; \tilde{\mathbb{r}}'_T \wedge \tilde{t}^r_T \;\tilde{\sqsubseteq}_t\; \tilde{t}^r{}'_T \wedge \tilde{t}^a_T \;\tilde{\sqsubseteq}_t\; \tilde{t}^a{}'_T)\end{gathered}$$

The monotonicity of γ_{reg}, γ_t and γ_{var} implies that $c \in \gamma_{conf}(\tilde{c}')$ as well. Thus, $\gamma_{conf}(\tilde{c}) \subseteq \gamma_{conf}(\tilde{c}')$ and the lemma holds. ◀

Now, the concretisation function handling sets of abstract configurations (the actual abstract domain for configurations), $\gamma_{pconf} : \mathcal{P}(\tilde{\mathbf{Conf}}) \to \mathcal{P}(\mathbf{Conf})$, is defined.

▶ **Definition 4.21** (Concretisation of a set of abstract configurations)**.**

$$\gamma_{pconf}(\tilde{C}) = \bigcup \{\gamma_{conf}(\tilde{c}) \mid \tilde{c} \in \tilde{C}\}$$ ◀

▶ **Lemma 4.22** (Monotonicity of γ_{pconf})**.** *The concretisation function $\gamma_{pconf} : \mathcal{P}(\tilde{\mathbf{Conf}}) \to \mathcal{P}(\mathbf{Conf})$ is monotone.* ◀

Proof. Trivial, since γ_{conf} is monotone (Lemma 4.20) and $\cup$ preserves monotonicity. ◀

The greatest lower bound operator for two abstract configurations $\tilde{\sqcap}_{conf}$ follows naturally using Definition 3.13.

▶ **Definition 4.23** (Greatest lower bound for two abstract configurations)**.**

$$\begin{cases}
\tilde{c} \,\tilde{\sqcap}_{conf}\, \tilde{\top}_{conf} = \tilde{\top}_{conf} \,\tilde{\sqcap}_{conf}\, \tilde{c} = \tilde{c} \\
\tilde{c} \,\tilde{\sqcap}_{conf}\, \tilde{\bot}_{conf} = \tilde{\bot}_{conf} \,\tilde{\sqcap}_{conf}\, \tilde{c} = \tilde{\bot}_{conf} \\
\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \,\tilde{\sqcap}_{conf} \\
\qquad \langle \{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^r{}'_T, \tilde{t}^a{}'_T) \mid T \in \mathbf{Thrd}_{\tilde{c}'}\}, \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}'\rangle = \\
\quad \begin{cases}
\langle \{(T, pc_T, \tilde{\mathbb{r}}_T \,\tilde{\sqcap}_{reg}\, \tilde{\mathbb{r}}'_T, \tilde{t}^r_T \,\tilde{\sqcap}_t\, \tilde{t}^r{}'_T, & \textbf{if } \mathbb{l} = \mathbb{l}' \wedge \mathbf{Thrd}_{\tilde{c}} = \mathbf{Thrd}_{\tilde{c}'} \wedge \\
\qquad \tilde{t}^a_T \,\tilde{\sqcap}_t\, \tilde{t}^a{}'_T \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, & \quad \forall T \in \mathbf{Thrd}_{\tilde{c}} : pc_T = pc'_T \\
\qquad \tilde{\mathbb{x}} \,\tilde{\sqcap}_{var}\, \tilde{\mathbb{x}}', \mathbb{l}, \tilde{t} \,\tilde{\sqcap}_t\, \tilde{t}'\rangle & \\
\tilde{\bot}_{conf} & \textbf{otherwise}
\end{cases}
\end{cases}$$ ◀

The least upper bound operator for two abstract configurations $\tilde{\sqcup}_{conf}$ follows naturally using Definition 3.14.

▶ **Definition 4.24** (Least upper bound for two abstract configurations)**.**

$$\begin{cases}
\tilde{c} \,\tilde{\sqcup}_{conf}\, \tilde{\top}_{conf} = \tilde{\top}_{conf} \,\tilde{\sqcup}_{conf}\, \tilde{c} = \tilde{\top}_{conf} \\
\tilde{c} \,\tilde{\sqcup}_{conf}\, \tilde{\bot}_{conf} = \tilde{\bot}_{conf} \,\tilde{\sqcup}_{conf}\, \tilde{c} = \tilde{c} \\
\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \,\tilde{\sqcup}_{conf} \\
\qquad \langle \{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^r{}'_T, \tilde{t}^a{}'_T) \mid T \in \mathbf{Thrd}_{\tilde{c}'}\}, \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}'\rangle = \\
\quad \begin{cases}
\langle \{(T, pc_T, \tilde{\mathbb{r}}_T \,\tilde{\sqcup}_{reg}\, \tilde{\mathbb{r}}'_T, \tilde{t}^r_T \,\tilde{\sqcup}_t\, \tilde{t}^r{}'_T, & \textbf{if } \mathbb{l} = \mathbb{l}' \wedge \mathbf{Thrd}_{\tilde{c}} = \mathbf{Thrd}_{\tilde{c}'} \wedge \\
\qquad \tilde{t}^a_T \,\tilde{\sqcup}_t\, \tilde{t}^a{}'_T \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, & \quad \forall T \in \mathbf{Thrd}_{\tilde{c}} : pc_T = pc'_T \\
\qquad \tilde{\mathbb{x}} \,\tilde{\sqcup}_{var}\, \tilde{\mathbb{x}}', \mathbb{l}, \tilde{t} \,\tilde{\sqcup}_t\, \tilde{t}'\rangle & \\
\tilde{\top}_{conf} & \textbf{otherwise}
\end{cases}
\end{cases}$$ ◀

Creating the abstraction function will require special care. This is due to the fact that any set of concrete configurations cannot be collected in one single abstract configuration. This is easy to see, e.g., by noticing that the program counters, pc, are used to determine unique program points and can thus not be abstracted (several concrete program points cannot be collected into one single abstract program point).

An abstraction function, $\alpha_{conf} : \mathbf{Conf} \to \tilde{\mathbf{Conf}}$, can be defined as:

▶ **Definition 4.25** (Abstraction of one concrete configuration)**.**

$$\begin{aligned}
&\alpha_{conf}(\langle \{(T, pc_T, \mathbb{r}_T, t^r_T, t^a_T) \mid T \in \mathbf{Thrd}\}, \mathbb{x}, \mathbb{l}, t\rangle) = \\
&\quad \langle \{(T, pc_T, \alpha_{reg}(\{\mathbb{r}_T\}), \alpha_t(\{t^r_T\}), \alpha_t(\{t^a_T\})) \mid T \in \mathbf{Thrd}\}, \alpha_{var}(\{\mathbb{x}\}), \mathbb{l}, \alpha_t(\{t\})\rangle
\end{aligned}$$ ◀

$\text{STM}(T, pc)$	$\langle pc', \tilde{\mathbb{r}}', \tilde{\mathbb{x}}', \tilde{\mathbb{l}}' \rangle$	Condition
$[\texttt{halt}]^{pc}$	$\langle pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$	–
$[\texttt{skip}]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$	–
$[r \texttt{ := } a]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}[r \mapsto \tilde{\mathcal{A}}[\![a]\!]\tilde{\mathbb{r}}], \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$	–
$[\texttt{load } r \texttt{ from } x]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}[r \mapsto \text{READ}(\tilde{\mathbb{x}}, x, T, \tilde{t})]\ \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$	–
$[\texttt{store } r \texttt{ to } x]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \text{WRITE}(T, \tilde{\mathbb{x}}, x, (\tilde{\mathbb{r}}\ r, \tilde{t})), \tilde{\mathbb{l}} \rangle$	–
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle pc+1, \tilde{\mathcal{BR}}[\![\texttt{!}b]\!]\tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$	$\tilde{\mathcal{BR}}[\![\texttt{!}b]\!]\tilde{\mathbb{r}} \neq \tilde{\bot}_{reg}$
$[\texttt{if } b \texttt{ goto } l]^{pc}$	$\langle l, \tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$	$\tilde{\mathcal{BR}}[\![b]\!]\tilde{\mathbb{r}} \neq \tilde{\bot}_{reg}$
$[\texttt{lock } lck]^{pc}$	$\langle pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$	$\text{OWN}(\tilde{\mathbb{l}}\ lck) \neq T$
$[\texttt{lock } lck]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}[lck \mapsto (locked, T)] \rangle$	$\text{OWN}(\tilde{\mathbb{l}}\ lck) = T$
$[\texttt{unlock } lck]^{pc}$	$\langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}[lck \mapsto (unlocked, \bot_{thrd})] \rangle$	–

Figure 11 Semantics of abstract axiom transitions: $\langle T, pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}, \tilde{t} \rangle \xrightarrow[ax]{\sim} \langle pc', \tilde{\mathbb{r}}', \tilde{\mathbb{x}}', \tilde{\mathbb{l}}' \rangle$

The abstraction function $\alpha_{pconf} : \mathcal{P}(\mathbf{Conf}) \to \mathcal{P}(\tilde{\mathbf{Conf}})$ is defined using Definition 3.15.

▶ **Definition 4.26** (Abstraction of a set of concrete configurations)**.**

$$\alpha_{pconf}(C) = \bigcap \{\tilde{C} \mid C \subseteq \gamma_{pconf}(\tilde{C})\}$$ ◀

▶ **Theorem 4.27** (Galois connection)**.** $\langle \alpha_{pconf} : \mathcal{P}(\mathbf{Conf}) \to \mathcal{P}(\tilde{\mathbf{Conf}}), \gamma_{pconf} : \mathcal{P}(\tilde{\mathbf{Conf}}) \to \mathcal{P}(\mathbf{Conf}) \rangle$ *is a Galois connection.* ◀

Proof. Using Theorems 3.6, 3.7, 3.8, 3.9, 3.10, 3.11 and 3.25, the result follows. ◀

The abstract transition rule for axiom statements in Figure 11 is a safe approximation of the rule in Figure 2, with respect to Definition 4.28.

▶ **Definition 4.28** (Safety of abstract axiom transition relations)**.** The transition relation $\xrightarrow[ax]{\sim}$ is a safe abstract approximation of $\xrightarrow[ax]{}$, with respect to γ^{ax}_{in} and γ^{ax}_{out}, iff:

$$\forall \tilde{c}^{ax}_{in} : \forall c^{ax}_{in} \in \gamma^{ax}_{in}(\tilde{c}^{ax}_{in}) : \exists \tilde{c}^{ax}_{out} : \exists c^{ax}_{out} \in \gamma^{ax}_{out}(\tilde{c}^{ax}_{out}) : (c^{ax}_{in} \xrightarrow[ax]{} c^{ax}_{out} \wedge \tilde{c}^{ax}_{in} \xrightarrow[ax]{\sim} \tilde{c}^{ax}_{out})$$ ◀

▶ **Theorem 4.29** (Safety of $\xrightarrow[ax]{\sim}$)**.** $\xrightarrow[ax]{\sim}$ *is a safe approximation of* $\xrightarrow[ax]{}$*, with respect to Definition 4.28.* ◀

Proof. This proof will be conducted by showing for each defined transition that it is safe according to Definition 4.28. Assume that $\tilde{c}^{ax}_{in} = \langle T, pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}}, \tilde{t} \rangle$ and $c^{ax}_{in} = \langle T, pc, \mathbb{r}, \mathbb{x}, \mathbb{l}, t \rangle$, such that $c^{ax}_{in} \in \gamma^{ax}_{in}(\tilde{c}^{ax}_{in})$.

1. Assume that $\text{STM}(T, pc) = [\texttt{halt}]^{pc}$. $\tilde{c}^{ax}_{out}$ is chosen so that $\tilde{c}^{ax}_{in} \xrightarrow[ax]{\sim} \tilde{c}^{ax}_{out}$, i.e., $\tilde{c}^{ax}_{out} = \langle pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$. From the concrete semantics, it must be that $c^{ax}_{in} \xrightarrow[ax]{} c^{ax}_{out}$, where $c^{ax}_{out} = \langle pc, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$. Thus, $c^{ax}_{out} \in \gamma^{ax}_{out}(\tilde{c}^{ax}_{out})$.
2. Assume that $\text{STM}(T, pc) = [\texttt{skip}]^{pc}$. $\tilde{c}^{ax}_{out}$ is chosen so that $\tilde{c}^{ax}_{in} \xrightarrow[ax]{\sim} \tilde{c}^{ax}_{out}$, i.e., $\tilde{c}^{ax}_{out} = \langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$. From the concrete semantics, we have that $c^{ax}_{in} \xrightarrow[ax]{} c^{ax}_{out}$, where $c^{ax}_{out} = \langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l} \rangle$. Thus, $c^{ax}_{out} \in \gamma^{ax}_{out}(\tilde{c}^{ax}_{out})$.
3. Assume that $\text{STM}(T, pc) = [r \texttt{ := } a]^{pc}$. Then $\tilde{c}^{ax}_{out}$ is chosen so that $\tilde{c}^{ax}_{in} \xrightarrow[ax]{\sim} \tilde{c}^{ax}_{out}$, i.e., $\tilde{c}^{ax}_{out} = \langle pc+1, \tilde{\mathbb{r}}[r \mapsto \tilde{\mathcal{A}}[\![a]\!]\tilde{\mathbb{r}}], \tilde{\mathbb{x}}, \tilde{\mathbb{l}} \rangle$. From the concrete semantics, we have that $c^{ax}_{in} \xrightarrow[ax]{} c^{ax}_{out}$, where $c^{ax}_{out} = \langle pc+1, \mathbb{r}[r \mapsto \mathcal{A}[\![a]\!]\mathbb{r}], \mathbb{x}, \mathbb{l} \rangle$. Since $\tilde{\mathcal{A}}$ is a safe approximation of $\mathcal{A}$ and γ_{reg} is monotone, $c^{ax}_{out} \in \gamma^{ax}_{out}(\tilde{c}^{ax}_{out})$.

$$\frac{\forall T \in \mathbf{Thrd}_{exe} : \langle T, pc_T, \tilde{\mathbb{r}}_T, \tilde{\mathbb{x}}, \mathbb{l}'', \tilde{t}_T^{a\prime}\rangle \xrightarrow[ax]{\sim} \langle pc'_T, \tilde{\mathbb{r}}'_T, \tilde{\mathbb{x}}'_T, \mathbb{l}'_T\rangle}{\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}_T^r, \tilde{t}_T^a) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \xrightarrow[prg]{\sim} \langle \{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}_T^{r\prime}, \tilde{t}_T^{a\prime}) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}'\rangle}$$

where

$$\tilde{t}_T^{r\prime} = \begin{cases} \textsc{AbsFinTime}\ (\langle \{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}_T^r, \tilde{t}_T^a) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle, T) & \textbf{if } \tilde{t} \tilde{\sqcap}_t \tilde{t}_T^a \neq \tilde{\bot}_t \\ \tilde{t}_T^r & \textbf{otherwise} \end{cases}$$

$$\tilde{t}' = \alpha_t(\{t_{min}, t_{max}\}) \qquad \textbf{where } t_{min} = \min\{\min(\gamma_t(\tilde{t}_T^a +_t \tilde{t}_T^{r\prime})) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}$$
$$t_{max} = \min\{\max(\gamma_t(\tilde{t}_T^a +_t \tilde{t}_T^{r\prime})) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}$$

$$\tilde{t}_T^{a\prime} = \begin{cases} \tilde{t}_T^a +_t \tilde{t}_T^{r\prime} & \textbf{if } \tilde{t}' \tilde{\sqcap}_t (\tilde{t}_T^a +_t \tilde{t}_T^{r\prime}) \neq \tilde{\bot}_t \\ \tilde{t}_T^a & \textbf{otherwise} \end{cases}$$

$\mathbf{Thrd}_{exe} = \{T \in \mathbf{Thrd}_{\tilde{c}} \mid \tilde{t}' \tilde{\sqcap}_t \tilde{t}_T^{a\prime} \neq \tilde{\bot}_t\}$

$(\tilde{\mathbb{x}}'\ x)\ T = (\tilde{\mathbb{x}}'_T\ x)\ T$

$\mathbb{l}''\ lck = \ldots$ (same as in Figure 4)

$\mathbb{l}'\ lck = \ldots$ (same as in Figure 4)

Figure 12 Semantics of abstract program transitions: $\langle \tilde{\mathbf{Ts}}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \xrightarrow[prg]{} \langle \tilde{\mathbf{Ts}}', \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}'\rangle$

4. Assume that $\textsc{stm}(T, pc) = [\texttt{load}\ r\ \texttt{from}\ x]^{pc}$. Then $\tilde{c}_{out}^{ax}$ is chosen so that $\tilde{c}_{in}^{ax} \xrightarrow[ax]{\sim} \tilde{c}_{out}^{ax}$, i.e., $\tilde{c}_{out}^{ax} = \langle pc+1, \tilde{\mathbb{r}}[r \mapsto \textsc{read}(\tilde{\mathbb{x}}, x, T, \tilde{t})], \tilde{\mathbb{x}}, \mathbb{l}\rangle$. From the concrete semantics, $c_{in}^{ax} \xrightarrow[ax]{} c_{out}^{ax}$, where $c_{out}^{ax} = \langle pc+1, \mathbb{r}[r \mapsto v], \mathbb{x}, \mathbb{l}\rangle$, for $\{(v, t')\} = \bigcup_{T' \in \mathbf{Thrd}}((\mathbb{x}\ x)\ T')$. Since $\textsc{read}$ returns a safe value (Lemma 4.14), $v \in \gamma_{int}(\textsc{read}(\tilde{\mathbb{x}}, x, T, \tilde{t}))$ and thus $c_{out}^{ax} \in \gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$.
5. Assume that $\textsc{stm}(T, pc) = [\texttt{store}\ r\ \texttt{to}\ x]^{pc}$. Then $\tilde{c}_{out}^{ax}$ is chosen so that $\tilde{c}_{in}^{ax} \xrightarrow[ax]{\sim} \tilde{c}_{out}^{ax}$, i.e., $\tilde{c}_{out}^{ax} = \langle pc+1, \tilde{\mathbb{r}}, \textsc{write}(T, \tilde{\mathbb{x}}, x, (\tilde{\mathbb{r}}\ r, \tilde{t})), \mathbb{l}\rangle$. From the concrete semantics, $c_{in}^{ax} \xrightarrow[ax]{} c_{out}^{ax}$, where $c_{out}^{ax} = \langle pc+1, \mathbb{r}, \mathbb{x}[x \mapsto (\mathbb{x}\ x)[T \mapsto \{(\mathbb{r}\ r, t)\}]], \mathbb{l}\rangle$. Since $\textsc{write}$ returns a safe abstract variable store (Lemma 4.13), $\mathbb{x}[x \mapsto (\mathbb{x}\ x)[T \mapsto \{(\mathbb{r}\ r, t)\}]] \in \gamma_{var}(\textsc{write}(T, \tilde{\mathbb{x}}, x, (\tilde{\mathbb{r}}\ r, \tilde{t})))$, and thus $c_{out}^{ax} \in \gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$.
6. Assume that $\textsc{stm}(T, pc) = [\texttt{if}\ b\ \texttt{goto}\ l]^{pc}$. Then two possible cases must be considered.
   a. In the first case, $\mathcal{B}[\![b]\!]\mathbb{r} = \texttt{true}$. This means that $c_{in}^{ax} \xrightarrow[ax]{} c_{out}^{ax}$, where $c_{out}^{ax} = \langle l, \mathbb{r}, \mathbb{x}, \mathbb{l}\rangle$. Now, $\tilde{c}_{out}^{ax}$ is chosen so that $\tilde{c}_{in}^{ax} \xrightarrow[ax]{\sim} \tilde{c}_{out}^{ax}$, i.e., $\tilde{c}_{out}^{ax} = \langle l, \tilde{\mathcal{B}}[\![b]\!]\tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}\rangle$. Since $\tilde{\mathcal{B}}[\![b]\!]\tilde{\mathbb{r}}$ is safe, $\gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$ contains, at least, all cases where $\mathcal{B}[\![b]\!]\mathbb{r} = \texttt{true}$. Thus, it must be the case that $c_{out}^{ax} \in \gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$.
   b. In the second case, $\mathcal{B}[\![b]\!]\mathbb{r} = \texttt{false}$. This means that $c_{in}^{ax} \xrightarrow[ax]{} c_{out}^{ax}$, where $c_{out}^{ax} = \langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l}\rangle$. Now, $\tilde{c}_{out}^{ax}$ is chosen so that $\tilde{c}_{in}^{ax} \xrightarrow[ax]{\sim} \tilde{c}_{out}^{ax}$, i.e., $\tilde{c}_{out}^{ax} = \langle pc+1, \tilde{\mathcal{B}}[\![!b]\!]\tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}\rangle$. Since $\tilde{\mathcal{B}}[\![!b]\!]\tilde{\mathbb{r}}$ is safe, $\gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$ contains, at least, all cases where $\mathcal{B}[\![b]\!]\mathbb{r} = \texttt{false}$. Thus, it must be the case that $c_{out}^{ax} \in \gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$.
7. Assume that $\textsc{stm}(T, pc) = [\texttt{lock}\ lck]^{pc}$. Then two possible cases must be considered.
   a. In the first case, $\textsc{own}(\mathbb{l}\ lck) \neq T$. This means that $c_{in}^{ax} \xrightarrow[ax]{} c_{out}^{ax}$, where $c_{out}^{ax} = \langle pc, \mathbb{r}, \mathbb{x}, \mathbb{l}\rangle$. Now, $\tilde{c}_{out}^{ax}$ is chosen so that $\tilde{c}_{in}^{ax} \xrightarrow[ax]{\sim} \tilde{c}_{out}^{ax}$, i.e., $\tilde{c}_{out}^{ax} = \langle pc, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}\rangle$. Thus, it must be the case that $c_{out}^{ax} \in \gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$.
   b. In the second case, $\textsc{own}(\mathbb{l}\ lck) = T$. This means that $c_{in}^{ax} \xrightarrow[ax]{} c_{out}^{ax}$, where $c_{out}^{ax} = \langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l}[lck \mapsto (locked, T)]\rangle$. Now, $\tilde{c}_{out}^{ax}$ is chosen so that $\tilde{c}_{in}^{ax} \xrightarrow[ax]{\sim} \tilde{c}_{out}^{ax}$, i.e., $\tilde{c}_{out}^{ax} = \langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}[lck \mapsto (locked, T)]\rangle$. Thus, it must be the case that $c_{out}^{ax} \in \gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$.
8. Assume that $\textsc{stm}(T, pc) = [\texttt{unlock}\ lck]^{pc}$. Then $\tilde{c}_{out}^{ax}$ is chosen so that $\tilde{c}_{in}^{ax} \xrightarrow[ax]{\sim} \tilde{c}_{out}^{ax}$, i.e., $\tilde{c}_{out}^{ax} = \langle pc+1, \tilde{\mathbb{r}}, \tilde{\mathbb{x}}, \mathbb{l}[lck \mapsto (unlocked, \bot_{thrd})]\rangle$. From the concrete semantics, we have that $c_{in}^{ax} \xrightarrow[ax]{} c_{out}^{ax}$, where $c_{out}^{ax} = \langle pc+1, \mathbb{r}, \mathbb{x}, \mathbb{l}[lck \mapsto (unlocked, \bot_{thrd})]\rangle$. Thus, $c_{out}^{ax} \in \gamma_{out}^{ax}(\tilde{c}_{out}^{ax})$. ◀

The abstract transition rule for program configurations in Figure 12 is an approximation of the concrete rule in Figure 4. The abstract rule now defines a window in time, $\tilde{t}'$, that determines which threads are included in $\mathbf{Thrd}_{exe}$. The window reaches from the earliest point in time when some thread might update its *pc*, to the earliest point in time when some *pc* must be updated. AbsFinTime is assumed to be a safe approximation of finTime.

The abstract rule in Figure 12 is a safe approximation of the concrete rule in Figure 4 only if some certain conditions are met. It is safe given that $|\mathbf{Thrd}_{\tilde{c}}| = 1$, or if a `load`-, `lock`- or `unlock`-statement is not executed by any thread in $\mathbf{Thrd}_{exe}$. This is easy to see since if these conditions are met, the threads in $\mathbf{Thrd}_{exe}$ execute independently from each other. If some thread in $\mathbf{Thrd}_{exe}$ would execute for example a `load`-statement, dependencies are introduced between the threads, and the read function could return a value for which all possible writes have not been taken into account. Let's assume that $\mathbf{Thrd}_{exe} = \{T_1, T_2\}$, $\text{STM}(T_1, pc_{T_1}) = [\texttt{load}\ r\ \texttt{from}\ x]^{pc_{T_1}}$, $\text{STM}(T_2, pc_{T_2}) = [\texttt{skip}]^{pc_{T_2}}$ and $\text{STM}(T_2, pc_{T_2} + 1) = [\texttt{store}\ r'\ \texttt{to}\ x]^{pc_{T_2}+1}$. When a transition occurs, the `load`- and `skip`-statements are considered. However, if the execution time of the `store`-statement (the abstract "point" in time when the thread's *pc* is updated) overlaps with the execution time of the `load`-statement, the resulting value of r in T_1 should be affected by the value of r' in T_2, but this will not be the case. A similar reasoning holds for `lock`- and `unlock`-statements.

4.3 Analysis by Abstract Execution

Since the abstract transition rule, $\xrightarrow[prg]{\sim}$, of Figure 12 is not safe, one cannot simply use fixpoint-iterations [5, 10] on the abstract semantic rules to find a safe approximation to the concrete program semantics. Instead, a worklist algorithm will be defined that uses $\xrightarrow[prg]{\sim}$ in a safe way and handles the unsafe cases explicitly. The function abstractExecution in Algorithm 4.9 defines such an algorithm; the '@' symbol is used for denoting two ways of expressing the same thing (c.f., the "read as" operator in Haskell). Given a configuration, $\tilde{c}$, and a timeout, $\tilde{t}_{to}$, the function explores all the possible abstract transitions, until only final (all threads are standing on a `halt`-statement) and timed-out (all threads will update their *pc*:s at a point in time succeeding $\tilde{t}_{to}$) configurations remain. The function returns a set containing all the final and timed-out configurations. If a configuration is not final or timed-out, a transition will be performed. The threads executing `load`-statements are extracted and handled separately. This is done by recursively using abstractExecution for each such thread to simulate how the rest of the threads in the configuration can affect the read value. When the effects have been derived, they are merged and put in the target register for the thread that issues the `load`-statement. Next, a new configuration, in which the `load`:s have been performed, is added to the worklist. trim, defined in Algorithm 4.10, is used to safely remove parts of the history from $\tilde{\mathbb{x}}$ that cannot affect a `load`-statement in any thread at time $\tilde{t}$ or in the future. This is to lower the space complexity of abstractExecution. Note that splitSet, as defined in Algorithm 4.11, is used to split a set of writes into two parts where the first part contains all writes that overlap in time with the given time, and the second part contains all other writes.

▶ **Lemma 4.30** (Safety of trim). *If $\tilde{\mathbb{x}}$ contains safe write history at time $\tilde{t}$ (see Definition 4.11), then so does* $\text{TRIM}(\tilde{\mathbb{x}}, \tilde{t})$. ◀

Proof. Given that $\tilde{\mathbb{x}}$ is safe, it must be shown that, for any variable x and any thread T, $((\text{TRIM}(\tilde{\mathbb{x}}, \tilde{t})\ x)\ T)$ contains at least

1. all writes, $(\tilde{v}, \tilde{t}')$, of $((\tilde{\mathbb{x}}\ x)\ T)$ such that $\tilde{t} \tilde{<}_t \tilde{t}'$,
2. all the writes, $(\tilde{v}, \tilde{t}')$, of $((\tilde{\mathbb{x}}\ x)\ T)$ such that $\tilde{t} \tilde{\sqcap}_t \tilde{t}' \neq \tilde{\perp}_t$, and

Algorithm 4.9 Abstract Execution

function ABSTRACTEXECUTION($\tilde{c}, \tilde{t}_{to}$)
  $workset \leftarrow \{\tilde{c}\}$
  $finalset \leftarrow \emptyset$
  repeat
    $\tilde{c}@\langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \leftarrow$ CHOOSE($workset$)
    $workset \leftarrow workset \setminus \{\tilde{c}\}$
    if ISTIMEOUT($\tilde{c}, \tilde{t}_{to}$) $\vee$ ISFINAL($\tilde{c}$) **then**
      $finalset \leftarrow finalset \cup \{\tilde{c}\}$
    else
      $\mathbf{Thrd}_{\mathtt{load}} \leftarrow$ LOADTHRD($\tilde{c}$)
      if $\mathbf{Thrd}_{\mathtt{load}} \neq \emptyset \wedge |\mathbf{Thrd}_{\tilde{c}}| > 1$ **then**
        for all $T' \in \mathbf{Thrd}_{\mathtt{load}}$ **do**
          $\tilde{t}^r_{T'}{}' \leftarrow$ ABSFINTIME($\tilde{c}$, T')
          $x \leftarrow$ GETVARLOAD(STM($T', pc_{T'}$))
          $r \leftarrow$ GETREGLOAD(STM($T', pc_{T'}$))
          $\tilde{v} \leftarrow \tilde{\perp}_{val}$
          $\tilde{c}' \leftarrow \langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}} \setminus \{T'\}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$
          $\tilde{C}^f_{T'} \leftarrow$ ABSTRACTEXECUTION($\tilde{c}', (\tilde{t}^a_{T'} +_t \tilde{t}^r_{T'}{}') \,\tilde{\sqcap}_t\, \tilde{t}_{to}$)
          for all $\langle \tilde{\mathbf{T}}\mathbf{s}, \tilde{\mathbb{x}}', \mathbb{l}', \tilde{t}'\rangle \in \tilde{C}^f_{T'}$ **do**
            $\tilde{v} \leftarrow \tilde{v} \,\tilde{\sqcup}_{val}$ READ($\tilde{\mathbb{x}}', x, T', \tilde{t}^a_{T'} +_t \tilde{t}^r_{T'}{}'$)
          end for
          $pc'_{T'} \leftarrow pc_{T'} + 1$
          $\tilde{\mathbb{r}}'_{T'}\ r' \leftarrow \begin{cases} \tilde{v} & \textbf{if } r = r' \\ \tilde{\mathbb{r}}_{T'}\ r' & \textbf{otherwise} \end{cases}$
        end for
        $\tilde{c}' \leftarrow \langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}} \setminus \mathbf{Thrd}_{\mathtt{load}}\} \cup$
                $\{(T, pc'_T, \tilde{\mathbb{r}}'_T, \tilde{t}^r_T{}', \tilde{t}^a_T +_t \tilde{t}^r_T{}') \mid T \in \mathbf{Thrd}_{\mathtt{load}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$
        $workset \leftarrow workset \cup \{\tilde{c}'\}$
      else
        $\tilde{C} \leftarrow \{\tilde{c}' \mid \tilde{c} \xrightarrow[prg]{\sim} \tilde{c}'\}$
        $\tilde{C}' \leftarrow \{\langle \tilde{\mathbf{T}}\mathbf{s}, \text{TRIM}(\tilde{\mathbb{x}}, \tilde{t}), \mathbb{l}, \tilde{t}\rangle \mid \langle \tilde{\mathbf{T}}\mathbf{s}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \in \tilde{C}\}$
        $workset \leftarrow workset \cup \tilde{C}'$
      end if
    end if
  until $workset = \emptyset$
  return $finalset$
end function

Algorithm 4.10 Trim Variable Store

```
1: function TRIM(x̃, t̃)
2:     for all x ∈ Var do
3:         for all T ∈ Thrd do
4:             FUTURE_T ← {(ṽ, t̃′) ∈ (x̃ x) T | t̃ ≲_t t̃′}
5:             (OL_T, NOL_T) ← SPLITSET((x̃ x) T, t̃)
6:             ((x̃′ x) T) ← NOL_T \ FUTURE_T
7:         end for
8:         t̃_mrw ← MOSTRECENTWRITETIME(x̃′ x)
9:         for all T ∈ Thrd do
10:            t̃_mrw_T ← MOSTRECENTWRITETIMETHREAD(OL′_T)
11:            W_T ← {(ṽ, t̃′) ∈ ((x̃′ x) T) | max(γ_t(t̃′)) = max(γ_t(t̃_mrw_T)) ∧ t̃_mrw ⊓̃_t t̃_mrw_T ≠ ⊥̃_t}
12:            ((x̃″ x) T) ← FUTURE_T ∪ OL_T ∪ W_T
13:        end for
14:    end for
15:    return x̃″
16: end function
```

Algorithm 4.11 Split Set of Writes

```
1: function SPLITSET(set, t̃)
2:     OL ← {(ṽ, t̃′) | (ṽ, t̃′) ∈ set ∧ t̃ ⊓̃_t t̃′ ≠ ⊥̃_t}
3:     NOL ← {(ṽ, t̃′) | (ṽ, t̃′) ∈ set ∧ t̃ ⊓̃_t t̃′ = ⊥̃_t}
4:     return (OL, NOL)
5: end function
```

3. the latest (most recent) write, $(\tilde{v}, \tilde{t}')$, such that $\tilde{t}' \lesssim_t \tilde{t}$.

For 1, it is easy to see that, for each variable and each thread, all future writes, i.e., writes, $(\tilde{v}, \tilde{t}')$, such that $\tilde{t} \lesssim_t \tilde{t}'$, are included in the resulting abstract variable store. These writes are included in the set $FUTURE_T$.

For 2, it is easy to see that for each variable and each thread, all writes $(\tilde{v}, \tilde{t}')$ such that $\tilde{t} \tilde{\sqcap}_t \tilde{t}' \neq \tilde{\bot}_t$ are included in the resulting abstract variable store. These writes are included in the set OL_T.

For 3, it is easy to see that the latest (most recent) write for each thread and variable among the writes in $NOL_T \setminus FUTURE_T$, i.e., the writes not included in OL_T or $FUTURE$, also are included in the resulting abstract variable store. These writes are included in the set W_T. ◀

CHOOSE returns one of the elements in the given non-empty set. ISFINAL, ISTIMEOUT, LOADTHRD, GETVARLOAD and GETREGLOAD are defined in Algorithm 4.12, 4.13, 4.14, 4.16 and 4.17, respectively. Note that EXETHRD is defined in Algorithm 4.15. ABSTRACTEXECUTION cannot, yet, safely analyse programs acting on locks. The algorithm will be extended with this ability (see Section 6).

Algorithm 4.12 Final Configuration

```
1: function ISFINAL(⟨{(T, pc_T, r̃_T, t̃^r_T, t̃^a_T) | T ∈ Thrd_c̃}, x̃, l, t̃⟩)
2:     return ∀T ∈ Thrd_c̃ : STM(T, pc_T) = [halt]^pc_T
3: end function
```

To derive the BCET and WCET of a program given an initial system state, ANALYSIS, defined in Algorithm 4.18, can be used. Note that LOCKS, defined in Algorithm 4.19,

Algorithm 4.13 Timeout

function ISTIMEOUT($\tilde{c}@\langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle, \tilde{t}_{to}$)
  for all $T \in \mathbf{Thrd}_{\tilde{c}}$ **do**
    $\tilde{t}^{r\,\prime}_T \leftarrow \begin{cases} \text{ABSFINTIME}(\tilde{c}, T) & \textbf{if } \tilde{t}^a_T \tilde{\sqcap}_t \tilde{t} \neq \tilde{\bot}_t \\ \tilde{t}^r_T & \textbf{otherwise} \end{cases}$
  end for
  return $\forall T \in \mathbf{Thrd}_{\tilde{c}} : (\text{STM}(T, pc_T) \neq [\texttt{halt}]^{pc_T} \Rightarrow \tilde{t}_{to} \tilde{<}_t (\tilde{t}^a_T +_t \tilde{t}^{r\,\prime}_T))$
end function

Algorithm 4.14 Find Threads with Load

function LOADTHRD($\tilde{c}@\langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$)
  $\mathbf{Thrd}_{exe} \leftarrow \text{EXETHRD}(\tilde{c})$
  $\mathbf{Thrd}_{\texttt{load}} \leftarrow \{T \in \mathbf{Thrd}_{exe} \mid \exists r \in \mathbf{Reg}_T, x \in \mathbf{Var} : \text{STM}(T, pc_T) = [\texttt{load } r \texttt{ from } x]^{pc_T}\}$
  return $\mathbf{Thrd}_{\texttt{load}}$
end function

Algorithm 4.15 Find Threads to Execute

function EXETHRD($\tilde{c}@\langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$)
  for all $T \in \mathbf{Thrd}_{\tilde{c}}$ **do**
    $\tilde{t}^{r\,\prime}_T \leftarrow \begin{cases} \text{ABSFINTIME}(\tilde{c}, T) & \textbf{if } \tilde{t} \tilde{\sqcap}_t \tilde{t}^a_T \neq \tilde{\bot}_t \\ \tilde{t}^r_T & \textbf{otherwise} \end{cases}$
    $t_{min} \leftarrow \min\{\min(\gamma_t(\tilde{t}^a_T +_t \tilde{t}^{r\,\prime}_T)) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}$
    $t_{max} \leftarrow \min\{\max(\gamma_t(\tilde{t}^a_T +_t \tilde{t}^{r\,\prime}_T)) \mid T \in \mathbf{Thrd}_{\tilde{c}}\}$
    $\tilde{t}' \leftarrow \alpha_t(\{t_{min}, t_{max}\})$
    $\tilde{t}^{a\,\prime}_T \leftarrow \begin{cases} \tilde{t}^a_T +_t \tilde{t}^{r\,\prime}_T & \textbf{if } \tilde{t}' \tilde{\sqcap}_t (\tilde{t}^a_T +_t \tilde{t}^{r\,\prime}_T) \neq \tilde{\bot}_t \\ \tilde{t}^a_T & \textbf{otherwise} \end{cases}$
  end for
  $\mathbf{Thrd}_{exe} \leftarrow \{T \in \mathbf{Thrd}_{\tilde{c}} \mid \tilde{t}' \tilde{\sqcap}_t \tilde{t}^{a\,\prime}_T \neq \tilde{\bot}_t\}$
  return $\mathbf{Thrd}_{exe}$
end function

Algorithm 4.16 Get Variable in Load

function GETVARLOAD($[\texttt{load } r \texttt{ from } x]^l$)
  return x
end function

Algorithm 4.17 Get Register in Load

function GETREGLOAD($[\texttt{load } r \texttt{ from } x]^l$)
  return r
end function

Algorithm 4.18 BCET/WCET Analysis

procedure ANALYSIS($\tilde{c}$)
  if LOCKS($\tilde{c}$) $\neq \emptyset$ **then**
    $BCET \leftarrow 0$
    $WCET \leftarrow \infty$
  else
    $finals \leftarrow$ ABSTRACTEXECUTION($\tilde{c}, \alpha_t(\{0, \infty\})$)
    $BCET \leftarrow \infty$
    $WCET \leftarrow -\infty$
    while $finals \neq \emptyset$ **do**
      $\tilde{c}@\langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle \leftarrow$ CHOOSE($finals$)
      $finals \leftarrow finals \setminus \{\tilde{c}\}$
      $BCET_{\tilde{c}} \leftarrow \max(\{\min(\gamma_t(\tilde{t}^a_T)) \mid T \in \mathbf{Thrd}\})$
      $WCET_{\tilde{c}} \leftarrow \max(\{\max(\gamma_t(\tilde{t}^a_T)) \mid T \in \mathbf{Thrd}\})$
      if $BCET > BCET_{\tilde{c}}$ **then**
        $BCET \leftarrow BCET_{\tilde{c}}$
      end if
      if $WCET < WCET_{\tilde{c}}$ **then**
        $WCET \leftarrow WCET_{\tilde{c}}$
      end if
    end while
  end if
end procedure

Algorithm 4.19 Get a Set of All Locks Used by the Program

function LOCKS($\langle\{(T, pc_T, \tilde{\mathbb{r}}_T, \tilde{t}^r_T, \tilde{t}^a_T) \mid T \in \mathbf{Thrd}\}, \tilde{\mathbb{x}}, \mathbb{l}, \tilde{t}\rangle$)
  Lck $\leftarrow \emptyset$
  for all $(s, N) \in \mathbf{Thrd}$ **do**
    for all $pc \in$ LABELS(s) **do**
      if STM($(s, N), pc$) = [`lock` lck]^{pc} **then**
        Lck $\leftarrow$ **Lck** $\cup \{lck\}$
      end if
    end for
  end for
end function

returns a set of all locks used by the program and that ANALYSIS evaluates the BCET to 0 and WCET to ∞ if this set is non-empty since ABSTRACTEXECUTION cannot be used to analyse programs acting on locks. However, if the program only issues `unlock`-statements, ABSTRACTEXECUTION can be safely used since this does not imply any synchronisation of the threads in the program.

Since the algorithms will be extended to handle programs acting on locks, full proofs of the correctness of the presented algorithms will not be provided. However, sketches for these proofs follow.

▶ **Lemma 4.31** ($\xrightarrow[prg]{\sim}$ is used in a safe way). *$\xrightarrow[prg]{\sim}$ is used by* ABSTRACTEXECUTION *in a safe way.* ◀

Proof sketch. The structure of the algorithm is such that whenever $|\mathbf{Thrd}_{\tilde{c}}| > 1$ and a `load`-statement is encountered, it is handled explicitly by the algorithm. Only when $|\mathbf{Thrd}_{\tilde{c}}| = 1$, $\xrightarrow[prg]{\sim}$ is used to handle the `load`-statement. Thus, the use of $\xrightarrow[prg]{\sim}$ is safe. ◀

▶ **Lemma 4.32** (Safety of `load`-statements). *A `load`-statement in some thread will see (at least) all the possible values of the read variable that are valid at the read-time, $\tilde{t}^a +_t \tilde{t}^{r\prime}$, given that the variable store contains safe write history (Definition 4.11) at $\tilde{t}^a +_t \tilde{t}^{r\prime}$.* ◀

Proof sketch. When a `load`-statement is handled explicitly by the algorithm, the thread executing the `load`-statement is "removed" from the configuration and all effects from the other threads that might occur before $\tilde{t}^a +_t \tilde{t}^{r\prime}$ expires are evaluated before reading the variable. When reading the variable, READ is used. Since READ is safe (Lemma 4.14) and the least upper bound of the read values are put in the register, the handling of `load`-statements is safe. ◀

▶ **Theorem 4.33** (Safety of ABSTRACTEXECUTION). *Assuming that* ABSFINTIME *is a safe approximation of* FINTIME, ABSTRACTEXECUTION *can be used to calculate a safe approximation of the execution-result of a program.* ◀

Proof sketch. The algorithm calculates all the possible resulting abstract configurations given an initial configuration, using $\xrightarrow[prg]{\sim}$ or explicitly whenever a `load`-statement is encountered. The resulting configurations are then added to a "work list" and later evaluated in the same way. When the work list is empty, i.e., all configurations are in a final or timed-out state, the algorithm returns. Thus, all possible executions are taken into account and since $\xrightarrow[prg]{\sim}$ is used in a safe way (Lemma 4.31) and the handling of `load`-statements is safe (Lemma 4.32), the result of executing a program is safely approximated. ◀

▶ **Theorem 4.34** (Safety of ANALYSIS). ANALYSIS *is safe, i.e., the calculated BCET and WCET are safe bounds on the concrete BCET and WCET, respectively.* ◀

Proof sketch. Since ABSTRACTEXECUTION, which is safe (Theorem 4.33), is used to calculate all the possible resulting configurations from executing a program, and ANALYSIS simply extracts the BCET and WCET from these configurations, ANALYSIS is safe. ◀

5 Example

In this section, the program in Figure 13 is analysed (the results of ABSFINTIME are given after the statements). Initially, let $\tilde{c} = \langle\{(T_1, 1, \tilde{\mathbb{r}}_{T_1}, [0,0], [0,0]), (T_2, 1, \tilde{\mathbb{r}}_{T_2}, [0,0], [0,0]), (T_3, 1, \tilde{\mathbb{r}}_{T_3}, [0,0], [0,0])\}, \tilde{\mathbb{x}}, \mathbb{l}, [0,0]\rangle$, where $\tilde{\mathbb{r}}_{T_3}\ \mathtt{r} = [2,4]$, $((\tilde{\mathbb{x}}\ \mathtt{x})\ T_2) = ((\tilde{\mathbb{x}}\ \mathtt{x})\ T_3) = \emptyset$ and

```
thread T_1:              thread T_2:              thread T_3:
[load r from x]1;[1,5]   [load r from y]1;[1,6]   [if r<=3 goto 3]1;[1,3]
[store r to y]2;[1,3]    [store r to z]2;[2,3]    [store r to x]2;[2,3]
[halt]3                  [halt]3                  [halt]3
```

Figure 13 Example program.

$((\tilde{\mathtt{x}}\ \mathtt{x})\ T_1) = \{([1,1],[0,0])\}$, $((\tilde{\mathtt{x}}\ \mathtt{y})\ T_1) = ((\tilde{\mathtt{x}}\ \mathtt{y})\ T_2) = \emptyset$ and $((\tilde{\mathtt{x}}\ \mathtt{y})\ T_3) = \{([5,5],[0,0])\}$, and $((\tilde{\mathtt{x}}\ \mathtt{z})\ T_2) = \emptyset$, is analysed. abstractExecution$(\tilde{c}, [0,\infty])$ is summarised in Figure 14.

The tuples in the chart represent program points, defined as $\langle pc_{T_1}, pc_{T_2}, pc_{T_3}\rangle$. As can be seen, for $\langle 1,1,1\rangle$, T_1 and T_2 both execute a `load`-statement. This means that two new instances of abstractExecution are created, one for each thread in $\mathbf{Thrd}_{\mathtt{load}}$. Within each of these instances, a new instance is created since one other thread also executes a `load`-statement. A '_' within the tuple indicates that the corresponding thread is removed from the configuration to evaluate the effects it might see. Next to each tuple and transition arrow, there is a comment stating what happens at the corresponding step. The found bounds on the BCET and WCET are 3 and 9, respectively. Note that absFinTime is assumed to be defined somewhere outside the scope of this paper.

6 Discussion & Future Work

The algorithm in Algorithm 4.9 is based on synchronously advancing the threads of a program between their respective program points. This, together with the defined abstract domain for variables, has the advantage that the analysis result will be the same as for the sequential case [7], when $P = T$. Another advantage is that the complexity of the algorithm becomes more dependent on the number of program points than on the timing behaviour of the program. To further reduce the time complexity of the algorithm, merging of configurations could be performed. Using the control flow graph (CFG) of the program, suitable merge-points within each thread can be found [6]. Typically, such points have multiple incoming edges.

A drawback for the algorithm in Algorithm 4.9 is that termination is not guaranteed if a program consists of *infinite* loops. This could be resolved by adjusting the initial timeout, though.

Our current focus is to extend the algorithm to support programs using locks and then to implement and evaluate it. Allowing the use of locks introduces a risk for deadlocks (both in the analysed program and thus the algorithm). However, deadlocks could easily be detected and handled by the algorithm, because all threads, not standing on a `halt`-statement, would be waiting to acquire a lock that is locked and not owned by themselves. Thus, this detection allows termination of the analysis (with a resulting WCET of ∞) even if deadlocks occur.

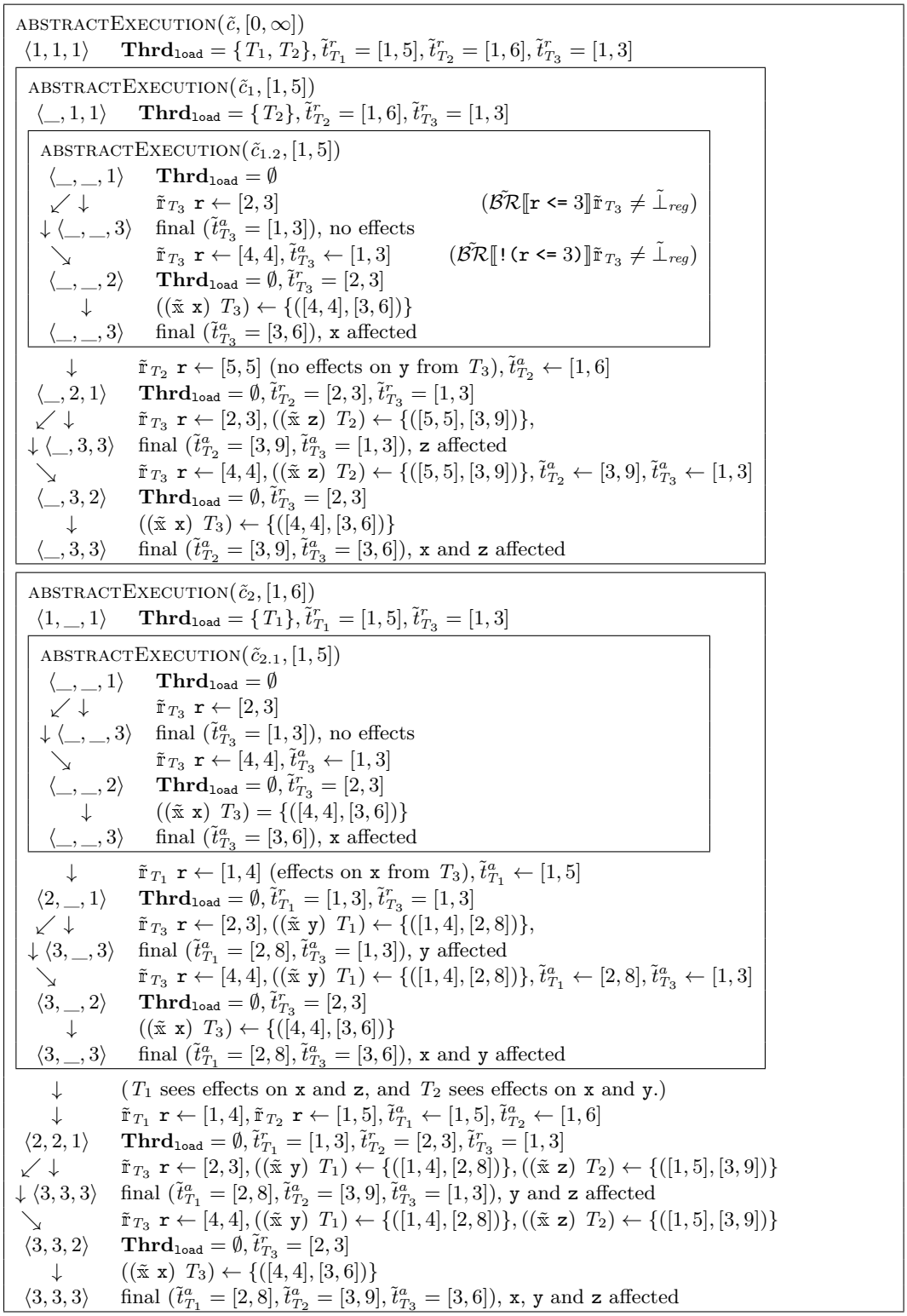


Figure 14 The steps taken by ABSTRACTEXECUTION when analysing the program in Figure 13.

References

1 Sudipta Chattopadhyay, C.-L. Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multi-core platforms. In *18th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'12)*, Beijing, China, April 2012.

2 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

3 Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving WCET bounds by abstract execution. In Chris Healy, editor, *Proc. 11th International Workshop on Worst-Case Execution Time Analysis (WCET'2011)*, Porto, Portugal, July 2011.

4 Andreas Ermedahl and Mikael Sjödin. Interval analysis of C-variables using abstract interpretation, 1996.

5 Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, May 2000.

6 Jan Gustafsson and Andreas Ermedahl. Merging techniques for faster derivation of WCET flow information using abstract execution. In Raimund Kirner, editor, *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'2008)*, Prague, Czech Republic, July 2008.

7 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Rio de Janeiro, Brazil, December 2006. IEEE Computer Society.

8 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 103–113, Brussels, Belgium, July 2010. OCG.

9 Mingsong Lv, Nan Guan, Wang Yi, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In Scott Brandt, editor, *Proc. 31th IEEE Real-Time Systems Symposium (RTSS'10)*, pages 339–349, San Diego, CA, December 2010. IEEE.

10 Flemming Nielson, Hanne Ries Nielson, and Chris Hankin. *Principles of Program Analysis, 2nd edition.* Springer, 2005. ISBN 3-540-65410-0.

11 Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 90–100, Brussels, Belgium, July 2010. OCG.

12 Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proc. 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 80–89, June 2008.