Automated logical verification based on trace abstractions

Nils Klarlund* Mogens Nielsen Kim Sunesen
BRICS†
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C.
{klarlund,mnielsen,ksunesen}@daimi.aau.dk

Abstract

We propose a practical framework for integrating the behavioral reasoning about distributed systems with model-checking methods.

Our proof methods are based on *trace abstractions*, which relate the behaviors of the program and the specification. We show that for finite-state systems such symbolic abstractions can be specified conveniently in a Monadic Second-Order Logic (M2L), which allows the concise expression of many temporal properties. Model-checking is then made possible by the reduction of non-determinism implied by the trace abstraction.

We outline how our method can be applied to a recent verification problem by Broy and Lamport. In an accompanying paper [11], we give a detailed account. The resulting complex temporal logic formulas are as long as 10-15 pages and are decided automatically within minutes.

*Author's current address: AT&T Bell Laboratories, Room 2C-410, 600 Mountain Ave., Murray Hill, NJ 07974; E-mail: klarlund@research.att.com.

†Basic Research in Computer Science, Centre of the Danish National Research Foundation .

1 Introduction

This paper is concerned with the specification and verification of distributed systems. Often, the relationship between a program and a specification is expressed in terms of a state-based refinement mapping, see [18] for a survey. Thus, when systems are specified by behavioral or temporal constraints, it is necessary first to find state-representations. In this process, important information may be lost or misconstrued.

In this paper, we exhibit a logic of *traces* (i.e. finite computation sequences) that allows compositional reasoning directly about behaviors. We formulate *trace abstractions* and their proof rules as an alternative to the use of refinement mappings for the verification of distributed systems.

Our main goal is to show that our method is useful in practice. Thus, use of our logic and proof rules must be supported by a decision procedure that will give answers to logical questions about the systems, such as "Does trace abstraction R show that program P implements specification S?"

To this end, we formulate a sound and complete verification method based on trace abstractions. We show that our method for finite-state systems can be formulated in a very succinct formalism: the *Monadic Second-order Logic* (M2L).

We address the important problem of relating a distributed program to a non-deterministic specification that also is a distributed system. Non-determinism arises when systems have alphabets that are partioned into observable and internal actions. Abstract-

ing away internal actions generally introduces nondeterminism.

Our contribution is to show an alternative to usual techniques, which tend to involve rather involved concepts such as *prophecy variables* or mappings to sets of sets of states. These can be replaced by behavioral predicates that need only to partially link the program and the specification. The remaining information is then calculated automata-theoretically by means of the subset construction.

We formulate a compositional rule to avoid the explicit construction of the global program space.

Using the Mona implementation of M2L, we have verified a recent verification problem by Broy and Lamport by transcribing several pages of informally stated temporal properties. The formulas resulting are decided in minutes despite their size (10^5 characters). We give here only an overview of our approach to the Broy and Lamport problem. A detailed treatement can be found in [11].

1.1 Relations to previous work

The systems we define are closely related to those described by Hoare in [9], where an alphabet Σ and a set of traces over Σ is associated with every process. We use a composition operator, similar to Hoare's parallel operator (|| [9]) forcing systems to synchronize on events or actions shared by both alphabets.

State mappings—one of the most advocated methods for proving refinement, see e.g. [17, 15, 16, 10] and for a survey [18]—were introduced as a way to avoid behavioral reasoning, often regarded as being too complex. The theory of state mappings is by now well-understood, but not simple, with the completeness results in [1, 12, 20]. In the finite state case, an important difference between the state mapping approach and ours is: in the traditional approaches, the mapping is to be exactly specified state by state, whereas in our approach the relation between behaviors may be specified partially leaving the rest to our verification tool.

In [2], Lamport and Abadi gave a proof rule for proving correctness of implementation of compound systems based on an assumption/guarantee method. A closed compound system is split into a number of open systems by factoring out dependencies as assumptions. Our rule is very different in that dependencies are reflected in a requirement about the relationship between trace abstractions for components.

The TLA formalism by Lamport [16] and the temporal logic of Manna and Pnueli [10] offer unified frameworks for specifying systems and state mappings, and for proving the correctness of implementation. Both logics are undecidable, but work has been done on establishing mechanical support, see [6, 7].

Clarke, Browne, and Kurshan [4] have applied model checking techniques to the language containement problem ($\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$), where M_1 and M_2 are ω-automata. They reduce the containment problem to a model-checking problem by forming a product of the automata and checking whether the product is a model for a certain CTL^* formula. The method is applicable to any common kind of ω-automata. Thus it deals with liveness properties unlike our method, which only deals with logic over finite prefixes. However, the method in [4] suffers from the restriction that M_2 be deterministic.

Kurshan, see [13], has devised an automata-theoretic framework for modeling and verifying synchronous transition systems. His use of homomorphisms allow complex properties to be reduced to ones that can be verified by means of model-checking.

Kurshan's methods were extended in [14] to the asynchronous *input/output automata* of [17]. There, Kurshan et al. give an account of interleaving composition in terms of conventional, synchronous automata. Our treatment of concurrency is similar in its use of stuttering for modeling asynchrony except that we do not consider fairness (which is a property of infinite sequences). A principal difference is that our proposal is based on comparing sequences of events, whereas the method of [14] is essentially state-based or event-based.

Binary Decision Diagrams (BDDs) are usually used in verification to compactify representations of state-spaces, see e.g. [5]. The Mona implementation [8] of a decision procedure for M2L uses BDDs to handle large alphabets.

1.2 Overview

In Section 2, we discuss our formal framework, which is based on an interleaving semantics of processes that

work in a global space of events. M2L is explained in Section 3. We show in Section 4 that with some additional concepts, it is possible to formulate the verification method of Section 2 in M2L. In Section 5, we explain the role of trace abstractions in our solution of the Broy and Lamport verification problem.

2 Traces and abstractions

We regard systems in a fairly standard way: they are devices that produce sequences of events that are either observable or internal. Systems exist in a universe. They can be composed and compared. Trace abstractions relate a program to a specification. These abstractions form a sound and complete verification method, and a simple decomposition rule is easy to formulate.

2.1 Systems and universes

A system A determines an alphabet Σ_A of *events*, which is partioned into *observable events* Σ_A^{Obs} and *internal events* Σ_A^{Int}. A *behavior* of A is a finite sequence over Σ_A. The system A also determines a prefix-closed language L_A of behaviors called *traces* of A. We write $A = (L_A, \Sigma_A^{Obs}, \Sigma_A^{Int})$. The *projection* π from a set Σ^* to a set Σ'^* $(\Sigma' \subseteq \Sigma)$ is the unique string homomorphism from Σ^* to Σ'^* given by $\pi(a) = a$, if a is in Σ', and $\pi(a) = \epsilon$ otherwise, where ϵ is the empty string. The *observable behaviors* of a system A, $Obs(A)$, are the projections on Σ_A^{Obs} of the traces of A, that is $Obs(A) = \{\pi(\alpha) \mid \alpha \in L_A\}$, where π is the projection from Σ_A^* onto $(\Sigma_A^{Obs})^*$.

A system A is thought of as existing in a *universe* which contains the systems with which it is composed and compared. The events possible in this universe constitute a global alphabet $\mathcal{U}$, which contains Σ_A and all other alphabets of interest. Moreover, $\mathcal{U}$ is assumed to contain the distinguished event τ, which is not in the alphabet of any system. The set $N_\Sigma(A)$ of *normalized traces* over an alphabet $\Sigma \supseteq \Sigma_A$ is the set $h^{-1}(L_A)$, where h is the projection from Σ^* onto Σ_A^*. Normalization plays an essential rôle when composing systems and when proving correctness of implementation of systems with internal events.

2.2 Composition

We say that systems A and B are *composable* if they do not disagree on the partition of events, that is, if no internal event of A is an observable event of B and vice versa, or symbolically, if $\Sigma_A^{Int} \cap \Sigma_B^{Obs} = \emptyset$ and $\Sigma_B^{Int} \cap \Sigma_A^{Obs} = \emptyset$. Given composable systems A and B, we define their *composition* $A \parallel B = (L_{A\parallel B}, \Sigma_{A\parallel B}^{Obs}, \Sigma_{A\parallel B}^{Int})$, where

- the set of observable events is the union of the sets of observable events of the components, that is, $\Sigma_{A\parallel B}^{Obs} = \Sigma_A^{Obs} \cup \Sigma_B^{Obs}$,
- the set of internal events is the union of the sets of internal events of the components, that is, $\Sigma_{A\parallel B}^{Int} = \Sigma_A^{Int} \cup \Sigma_B^{Int}$, and
- the set of traces is the intersection of the sets of normalized traces with respect to the alphabet $\Sigma_{A\parallel B}$, i.e. $L_{A\parallel B} = N_{\Sigma_{A\parallel B}}(A) \cap N_{\Sigma_{A\parallel B}}(B)$.

(Note that the restriction above for composability ensures that $A \parallel B$ has also disjoint observable and internal events.)

A trace of $A \parallel B$ is the interleaving of a trace of A with a trace of B in which common events are synchronized. The projection of a trace of $A \parallel B$ onto the alphabet of any of the components is a trace of the component. Composition is commutative, idempotent, and associative, and extends straightforwardly to any number n of composable systems A_i. We write $A_1 \parallel \ldots \parallel A_n$ or just $\parallel A_i$.

Example 2.1 To make the concepts clearer, we show how to present the well-known scheduler [19] of Milner in terms of our systems. The distributed scheduler is based on passing a token consecutively between a number of computing agents. We consider a three-agent version of the scheduler. The ith agent S_i performs observable events a_i and b_i to indicate the begining and the end of computing, respectively, and it synchronizes with its neighbor agents by interacting on the internal events c_i and $c_{i\ominus 1}$, where i is 0, 1, or 2, and $\ominus$ is subtraction modulo 3.

For a regular expression r, we denote by $\mathcal{L}^{Pre}(r)$ the regular language obtained by taking the prefix-closure of the language associated with r. Thus the agents may be described by:

$$S_0 = (\mathcal{L}^{Pre}((a_0c_0(b_0c_2 + c_2b_0))^*),\ \{a_0, b_0\}, \{c_0, c_2\}),$$
$$S_1 = (\mathcal{L}^{Pre}(c_0(a_1c_1(b_1c_0 + c_0b_1))^*), \{a_1, b_1\}, \{c_0, c_1\}),$$
$$S_2 = (\mathcal{L}^{Pre}(c_1(a_2c_2(b_2c_1 + c_1b_2))^*), \{a_2, b_2\}, \{c_1, c_2\})$$

The scheduler is defined in terms of the compound system:

$$S \quad = \quad S_0 \parallel S_1 \parallel S_2$$

where the set of observable events then consists of the a_i's and b_i's.

2.3 Implementation

We say that systems A and B are *comparable* if they have the same set of observable events Σ^{Obs}, that is, $\Sigma^{Obs} = \Sigma_A^{Obs} = \Sigma_B^{Obs}$. In the following, A and B denote comparable systems and π denotes the projection from $\mathcal{U}^*$ onto $(\Sigma^{Obs})^*$.

Definition 2.1 *A implements B* if and only if $Obs(A) \subseteq Obs(B)$.

Example 2.2 Another way of defining a scheduler is to use a central agent C. The ith agent still performs observable events a_i and b_i but now synchronizes with the agent C by interacting on the internal event d_i. The agents are given by the systems

$$C \quad = \quad (\mathcal{L}^{Pre}((d_0d_0d_1d_1d_2d_2)^*), \emptyset, \{d_0, d_1, d_2\}),$$
$$P_i \quad = \quad (\mathcal{L}^{Pre}((d_ia_id_ib_i)^*), \{a_i, b_i\}, \{d_i\}),\ i = 0, 1, 2$$

and the scheduler is defined by the compound system:

$$P \quad = \quad P_0 \parallel P_1 \parallel P_2 \parallel C$$

where the observable events are the a_i's and b_i's and internal events are the d_i's.

The systems S and P may be seen as existing in the universe $\mathcal{U} = \{a_i, b_i, c_i, d_i, \tau \mid i = 0, 1, 2\}$ and are clearly comparable. The reader may convince himself that P implements S, but in general this not an easy task.

2.4 Relational trace abstractions

A *trace abstraction* is a relation on traces preserving observable behaviors.

Definition 2.2 A trace abstraction $\mathcal{R}$ from A to B is a relation on $\mathcal{U}^* \times \mathcal{U}^*$ such that

1. If $\alpha \mathcal{R} \beta$ then $\pi(\alpha) = \pi(\beta)$
2. $N_{\mathcal{U}}(A) \subseteq dom\ \mathcal{R}$
3. $rng\ \mathcal{R} \subseteq N_{\mathcal{U}}(B)$

The first condition states that any pair of related traces must agree on observable events. The second and third condition require that any normalized trace of A should be related to some normalized trace of B, and only to normalized traces of B. The use of trace abstractions forms a sound and complete method in the sense that there exists a trace abstraction from A to B if and only if A implements B.

Theorem 2.1 There exists a trace abstraction from A to B if and only if A implements B.

We would like to prove that a compound system $\parallel A_i$ implements another compound system $\parallel B_i$ by exhibiting trace abstractions $\mathcal{R}_i$ from A_i to B_i. A simple extra condition is neeeded for this to work:

Theorem 2.2 Let A_i and B_i be pairwise comparable systems forming the compound systems $\|A_i$ and $\|B_i$. If

$$\mathcal{R}_i \text{ is a trace abstraction from } A_i \text{ to } B_i \qquad (1)$$
$$\textstyle\bigcap_i dom\ \mathcal{R}_i \ \subseteq \ dom\ \bigcap_i \mathcal{R}_i \qquad (2)$$

then

$$\|A_i \text{ implements } \|B_i$$

Intuitively, the extra condition (2), which we call the *compatibility requirement*, ensures that the choices defined by the trace abstractions can be made to agree on internal events.

Due to the possibility of non-trivial interference on internal events among the component systems, the first premise alone of the composition rule is not sufficient to ensure the conclusion. Consider e.g. the following systems

$$A_1 = (\{a\}^*, \{a\}, \emptyset), \qquad B_1 = (\{ac\}^*\{\epsilon, a\}, \{a\}, \{c\})$$
$$A_2 = (\{b\}^*, \{b\}, \emptyset), \qquad B_2 = (\{bc\}^*\{\epsilon, b\}, \{b\}, \{c\})$$

$Obs(A_i) \subseteq Obs(B_i)$, but $Obs(A_1 \parallel A_2) \not\subseteq Obs(B_1 \parallel B_2)$, since $aa \in Obs(A_1 \parallel A_2)$, but $aa \notin Obs(B_1 \parallel B_2)$.

The next example illustrates that even when significant internal interaction exists among the components, the decomposition theorem may be applied.

Example 2.3 Consider the schedulers from before. For each $i = 0, 1, 2$, let χ_i be the string homomorphism from $\mathcal{U}^*$ to $\mathcal{U}^*$ mapping every string α into a string identical to α except that every occurrence of c_i is erased and every even occurrence of d_i is replaced by c_i. Formally, $\chi_i(\epsilon) = \epsilon$ and for $\alpha \in \mathcal{U}^*$ and $u \in \mathcal{U}$,

$$\chi_i(\alpha u) = \begin{cases} \chi_i(\alpha)c_i & \text{if } u = d_i \text{ and} \\ & \text{the number of } d_i\text{s in } \alpha \text{ is odd} \\ \chi_i(\alpha) & \text{if } u = c_i \\ \chi_i(\alpha)u & \text{otherwise} \end{cases}$$

Let $\chi = \chi_0 \circ \chi_1 \circ \chi_2$. It is not hard to check that the relations $\mathcal{R}_i = \{(\alpha, \chi(\alpha)) \mid \chi(\alpha) \in N_{\mathcal{U}}(S_i)\}$ are trace abstractions from $P_i \parallel C$ to S_i, respectively. (Requirements 1. and 3. are satisfied by definition. To see that 2. holds, we consider some $\alpha \in N_{\mathcal{U}}(P_i \parallel C)$ and argue that $\chi(\alpha) \in N_{\mathcal{U}}(S_i)$.) Also, it is not hard to see that $\bigcap_i dom\, \mathcal{R}_i \subseteq dom \bigcap_i \mathcal{R}_i$. (For each α there is exactly one $\chi(\alpha)$.) Hence by Theorem 2.2, it follows that $(P_0 \parallel C) \parallel (P_1 \parallel C) \parallel (P_2 \parallel C)$ implements S and therefore that P implements S.

An almost trivial observation is:

Corollary 2.1 If additionally the components of the specification are non-interfering on internal events, that is, $\Sigma^{Int}_{B_i} \cap \Sigma^{Int}_{B_j} = \emptyset$, for every $i \neq j$, then A_i implements B_i implies $\|A_i$ implements $\|B_i$.

3 Monadic second-order logic on strings

The logical language we use is the monadic second-order logic (M2L) on strings, where a closed formula is interpreted relative to a natural number n (the *length*). First-order variables $p, q, \ldots$ range over the set $\{0, \ldots, n-1\}$ (the set of *positions*), and second-order variables $P, Q, \ldots, P_1, P_2, \ldots$ range over subsets of $\{0, \ldots, n-1\}$. Atomic formulas are of the form $p = q$, $p = q + 1$, $p < q$ and $q \in P$. Formulas are constructed in the standard way from atomic formulas by means of the Boolean connectives $\neg, \wedge, \vee, \Rightarrow$ and $\Leftrightarrow$, and first and second-order quantifiers $\forall$ and $\exists$. We adopt the standard notation of writing $\phi(P_1, \ldots, P_k, p_1, \ldots, p_l)$ to denote an open formula ϕ whose free variables are among $P_1, \ldots, P_k, p_1, \ldots, p_l$. Let 0 and \$ be the M2L definable constants denoting the positions 0 and $n-1$, respectively. The expressive power of M2L is illustrated by the formula

$$\exists P.0 \in P \wedge (\forall p.p < \$ \Rightarrow (p \in P \Leftrightarrow p + 1 \notin P))$$

which defines the set of even numbers. A second-order variable P can be seen as denoting a string of bits $b_0 \ldots b_{n-1}$ such that $b_i = 1$ if and only if $i \in P$. This leads to a natural way of associating a language $L(\phi)$ over $\Sigma = \mathbb{B}^m$ of satisfying interpretations to an open formula $\phi(P_1, \ldots, P_m)$ having only second-order variables occurring free ($\mathbb{B}$ denotes the set $\{0, 1\}$). As an example, consider the formula $\phi \equiv \forall p.p \in P_1 \Leftrightarrow p \notin P_2$. Then $L(\phi)$ is a language over the alphabet $\Sigma = \mathbb{B}^2$, where each $(b_1, b_2) \in \mathbb{B}^2$ denotes the membership status of the current position relative to P_1 and P_2. For example, writing the tuples as columns, we have

$$\begin{smallmatrix} P_1\colon 11010 \\ P_2\colon 00101 \end{smallmatrix} \in L(\phi) \text{ and } \begin{smallmatrix} P_1\colon 11010 \\ P_2\colon 01000 \end{smallmatrix} \notin L(\phi)$$

Any language defined by a M2L formula is regular and conversely any regular language can be defined by a M2L formula. Given a formula ϕ, a minimal finite automaton accepting $L(\phi)$ can effectively be constructed using the standard operations of complementation, product, subset construction, and projection. In particular, the existential quantifier becomes associated with a subset construction—and a potential exponential blow-up in the number of states. The construction of automata constitutes a decision procedure for M2L, since ϕ is a tautology if and only if $L(\phi)$ is the set of all strings. In case ϕ is not a tautology, a witness in terms of a minimal interpretation falsifying ϕ can be derived from the minimum deterministic automaton recognizing $L(\phi)$. We use the tool Mona [8], which implements the decision procedure and the counter-example facility.

4 The finite state case

We now restrict attention to systems with regular trace languages. We show for a large class of finite-state systems that trace abstractions definable by regular languages constitute a complete method for proving the implementation property.

Given strings $\alpha = \alpha_0 \ldots \alpha_n \in \Sigma_1^*$ and $\beta = \beta_0 \ldots \beta_n \in \Sigma_2^*$, we write $\alpha^\wedge\beta$ for the string $(\alpha_0, \beta_0) \ldots (\alpha_n, \beta_n) \in (\Sigma_1 \times \Sigma_2)^*$. Every language $L_\mathcal{R}$ over a product alphabet $\Sigma_1 \times \Sigma_2$ has a canonical embedding as a relation $\mathcal{R}_L \subseteq \Sigma_1^* \times \Sigma_2^*$ on strings of equal length given by $\alpha^\wedge\beta \in L_\mathcal{R} \stackrel{\text{def}}{\Leftrightarrow} \alpha \mathcal{R}_L \beta$. Hence in the following we shall use the two representations interchangeably. Accordingly, we say that a trace abstraction is *regular* if it is the embedding of a regular language over $\mathcal{U} \times \mathcal{U}$.

Not all trace abstractions between finite-state systems are regular, since there may be an unbounded number of internal events between pairs of corresponding observable events. The next definition is an essential step towards the identification of regular trace abstractions.

Definition 4.1 Given a subset Σ' of Σ, we say that strings $\alpha, \beta \in \Sigma^*$ are Σ'*–synchronized* if they are of equal length and if whenever the ith position in α contains a letter in Σ' then the ith position in β contains the same letter, and vice versa.

Definition 4.2 Let $\hat{\mathcal{R}}$ be the language over $\mathcal{U} \times \mathcal{U}$ given by $\alpha^\wedge\beta \in \hat{\mathcal{R}}$ if and only if

$$\beta \in N_\mathcal{U}(B) \text{ and } \alpha, \beta \text{ are } \Sigma^{Obs}\text{-synchronized}$$

Since $N_\mathcal{U}(B)$ is a regular language (by assumption of this Section), so is $\hat{\mathcal{R}}$. The next proposition gives a sufficient condition for $\hat{\mathcal{R}}$ and any regular subset of $\hat{\mathcal{R}}$ to be a trace abstraction. We return to the significance of the last part when dealing with automated proofs.

Proposition 4.1 If $N_\mathcal{U}(A) \subseteq dom\, \hat{\mathcal{R}}$ then $\hat{\mathcal{R}}$ is a regular trace abstraction from A to B. Moreover in general, for any regular language $\mathcal{C} \subseteq (\mathcal{U} \times \mathcal{U})^*$, if $N_\mathcal{U}(A) \subseteq dom\, \hat{\mathcal{R}} \cap \mathcal{C}$, then $\hat{\mathcal{R}} \cap \mathcal{C}$ is a regular trace abstraction from A to B.

It is not hard to see that if $\hat{\mathcal{R}}$ is a regular trace abstraction, then it is the largest such relating Σ^{Obs}-synchronized traces. In this case, we denote $\hat{\mathcal{R}}$ the *canonical* trace abstraction.

Non-regularity of trace abstractions occurs if for example there are arbitrarily many non-observable events between any two observable events. However, it may also happen that a behavior of the program may have too few internal events between two observable events in the sense that any behavior of the specification with the same observable behavior may require more internal events. We next give a precise definition of this phenomenon. Let π_A and π_B be the projections from Σ_A^* and Σ_B^*, respectively, onto $(\Sigma^{Obs})^*$.

Definition 4.3 A trace $\alpha \in L_A$ is *internally finer* than a trace $\beta \in L_B$ if $\pi_A(\alpha) = \pi_B(\beta)$, and for all $e, e' \in \Sigma^{Obs}, u \in (\Sigma_A^{Int})^*, v \in (\Sigma_B^{Int})^*, \alpha_1, \alpha_2 \in \Sigma_A^*$ and $\beta_1, \beta_2 \in \Sigma_B^*$, such that $\pi_A(\alpha_1) = \pi_B(\beta_1)$

$$\left.\begin{array}{lll} \alpha = \alpha_1 e u e' \alpha_2 & \wedge & \beta = \beta_1 e v e' \beta_2 \\ \vee & & \\ \alpha = u e' \alpha_2 & \wedge & \beta = v e' \beta_2 \end{array}\right\} \Rightarrow |u| \geq |v|$$

A system A is internally finer than a system B if for any trace α of A such that $\pi_A(\alpha) \in Obs(B)$, there exists a trace β of B such that α is internally finer than β.

Consider the scheduler example. System P is internally finer than S, whereas the converse is not true. We restate the soundness and completeness result from the general case for a constrained class of systems and regular trace abstractions.

Theorem 4.1 Assume that A is internally finer than B. There exists a canonical trace abstraction from A to B if and only if A implements B.

The restriction on programs to be internally finer than their specifications can be overcomed simply by adding more internal behavior to the program. More precisely, given systems A and B there always exists a system A' such that A and A' have the same observable behaviors, that is, $Obs(A) = Obs(A')$, and such that A' is internally finer than B. E.g. using $S_0' = (\mathcal{L}^{Pre}((d_0 a_0 d_0 c_0 (b_0 c_2 + c_2 b_0))^*), \{a_0, b_0\}, \{c_0, c_2, d_0\})$ instead of S_0 and with similar changes using S_1' and S_2' for S_1 and S_2, respectively, we have that $S' = S_0' \parallel S_1' \parallel S_2'$ is internally finer than P and that $Obs(S) = Obs(S')$.

4.1 A uniform logical framework

In the finite setting, reasoning about systems can conveniently be expressed in M2L. Let $\mathcal{U} = \mathbb{B}^m$ be the universe, where m is a natural number. Any behavior α over $\mathcal{U}$ can be viewed as an interpretation of a sequence of second-order variables $U_1^\alpha, \ldots, U_m^\alpha$. So behaviors over, say, 1024 different events may be coded using just 10 variables.

We use for each event $\sigma = (b_1, \ldots, b_m) \in \mathcal{U}$ and α the notation $\alpha(t) = \sigma$ for the M2L predicate

$$(\bigwedge_{b_i=1} t \in U_i^\alpha) \ \wedge (\bigwedge_{b_i=0} t \notin U_i^\alpha),$$

which states that the behavior denoted by α has a σ event in the position denoted by t. A system $A = (L_A, \Sigma_A^{Obs}, \Sigma_A^{Int})$ is represented by a triple

$$A = (\phi_A, \phi_A^{Obs}, \phi_A^{Int})$$

of formulas defining the normalized traces of the system, $\phi_A(\alpha)$, the observable events, $\phi_A^{Obs}(\alpha, t)$, and the internal events, $\phi_A^{Int}(\alpha, t)$. That is, $N_{\mathcal{U}}(A) = L(\phi_A)$ and $\phi_A^{Obs}(\alpha, t))$ and $\phi_A^{Int}(\alpha, t)$ are predicates that are true if and only if the position denoted by t in the behavior denoted by α is an element of Σ_A^{Obs} and Σ_A^{Int}, respectively. Given composable systems A and B, composition is represented by

$$A \parallel B = (\phi_A \wedge \phi_B, \phi_A^{Obs} \vee \phi_B^{Obs}, \phi_A^{Int} \vee \phi_B^{Int}).$$

We have that $L(\phi_A \wedge \phi_B) = L(\phi_A) \cap L(\phi_B) = N_{\mathcal{U}}(A \parallel B)$ and that $\phi_A^{Obs} \vee \phi_B^{Obs}$ and $\phi_A^{Int} \vee \phi_B^{Int}$ defines the union of the observable and the internal events, respectively. Let now behavior β be represented by $U_1^\beta, \ldots, U_m^\beta$. The property that behaviors α and β in $\mathcal{U}^*$ are Σ^{Obs}-synchronized is expressed by predicate $\phi_{A,B}^{Obs}(\alpha, \beta)$ defined by

$$\forall t : (\phi_A^{Obs}(\alpha, t) \vee \phi_B^{Obs}(\alpha, t)) \Rightarrow \alpha(t) = \beta(t).$$

The canonical trace abstraction $\hat{\mathcal{R}}$ of Definition 4.2 is defined by

$$\hat{\mathcal{R}}_{A,B}(\alpha, \beta) \quad \stackrel{\text{def}}{\equiv} \quad \phi_B(\beta) \ \wedge \ \phi_{A,B}^{Obs}(\alpha, \beta).$$

By Proposition 4.1 and Theorem 4.1, the implementation property is implied by $N_{\mathcal{U}}(A) \subseteq dom\, \hat{\mathcal{R}}$ and hence by the validity of

$$\phi_A(\alpha) \quad \Rightarrow \quad \exists \beta : \hat{\mathcal{R}}_{A,B}(\alpha, \beta), \tag{3}$$

where $\exists \beta$ is defined as $\exists U_1^\beta . \cdots \exists U_m^\beta$. Let $\mathcal{R}_i(\alpha, \beta) \stackrel{\text{def}}{\equiv} \hat{\mathcal{R}}_{A_i,B_i}(\alpha, \beta) \wedge \psi_i(\alpha, \beta)$. The premises of the decomposition rule in Theorem 2.2 are expressed by

$$\bigwedge_i (\phi_{A_i}(\alpha) \quad \Rightarrow \quad \exists \beta : \mathcal{R}_i(\alpha, \beta)) \tag{4}$$

$$\bigwedge_i \exists \beta_i : \mathcal{R}_i(\alpha, \beta_i) \quad \Rightarrow \quad \exists \beta : \bigwedge_i \mathcal{R}_i(\alpha, \beta). \tag{5}$$

To express the premise of Corollary 2.1 simply replace equation (5) above by

$$\bigwedge_{i \neq j} \forall t : \phi_{B_i}^{Int}(\alpha, t) \Rightarrow \neg \phi_{B_j}^{Obs}(\alpha, t).$$

Also, properties like composability and comparability can be expressed. The former by

$$\begin{aligned} \forall t : \quad & (\phi_A^{Int}(\alpha, t) \Rightarrow \neg \phi_B^{Obs}(\alpha, t)) \ \wedge \\ & (\phi_B^{Int}(\alpha, t) \Rightarrow \neg \phi_A^{Obs}(\alpha, t)) \end{aligned}$$

and the latter by

$$\forall t : \phi_A^{Obs}(\alpha, t) \Leftrightarrow \phi_B^{Obs}(\alpha, t).$$

In general, M2L is a very flexible logical language making it easy to write tense time and interval temporal logic operators in a straightforward manner. As examples, consider the past operator $\phi_{\sigma,\mu}^{Before}(\alpha)$ defined by

$$\forall t_1 . \alpha(t_1) = \mu \Rightarrow \exists t_0 . t_0 < t_1 \ \wedge \ \alpha(t_0) = \sigma,$$

and the interval operator $\phi_\sigma^{Between}(\alpha, t_1, t_2)$

$$\exists t . t_1 < t < t_2 \ \wedge \ \alpha(t) = \sigma.$$

4.2 Automated proofs

Formulas (3), (4), and (5) are potentially very difficult, since they involve quantification over behaviors, that is, over m second-order variables. Each quantification can lead to an exponential blow-up. But if A has much internal behavior, then it seems reasonable to use a more clever trace abstraction guided by A's internal events. In fact, it must be suspected that it is inappropriate that the definition of $\hat{\mathcal{R}}$ does not involve A at all.

The canonical trace abstraction can be constrained by adding more precise information about the connection between the internal behavior of system A and B. This may reduce the blow-up—or even avoid it in the case a functional regular trace abstraction is formulated.

We next turn to a substantial verification problem to illustrate our technique.

5 A specification problem

In this section, we consider the problem proposed by Broy and Lamport [3]. The first part of [3] calls for a specification of a reactive system consisting of a number of sequential processes issuing blocking read and write calls to a memory server. The memory server maintains its memory by performing special atomic reads and writes whenever requested to do so by read and write calls. Depending on the success of atomic reads and writes, return events contain the answers to read and write calls. The memory must be able to handle several calls (from different processes) concurrently.

The second part of [3] calls for an implementation based on a remote procedure call (rpc) protocol. The protocol involves a local and a remote party. Calls received locally are forwarded to the remote site, where they are executed. The resulting return events are propagated back to the local site. Altogether, we deal here with four levels of calls and returns.

The goal of [3] is now to verify that every observable trace of the implementation (where atomic read and writes and the remote events are abstracted away) is an observable trace of the specification.

The full informal description [3] includes many technical complications concerning the parameters passed and different kinds of erroneous behaviors. A detailed presentation of our solution can be found in [11].

In performing the verifications, we have limited ourselves to finite domains. We have chosen to have two locations, two kinds of values, two kinds of flags, and two process identities (in addition to the memory process). The resulting program has approximately a hundred thousand states and the specification approximately a thousand states. The systems allow thousands of different events. The systems are modelled as deterministic automata. The full specification amounts to 10-15 pages of M2L formulas (written in a macro language).

The aspect that we are interested in here is the use of trace abstractions. Without going into any further details, we assume that the M2L formulas $\phi_{P_1} \wedge \phi_{P_2}$ and $\phi_{S_1} \wedge \phi_{S_2}$ define the implementation and the specification, respectively, of our solution (each conjunct specifies the behavior of one process). The universe $\mathcal{U}$ consists of τ and a number of parameterized events: *rd, wrt, rtn, atmrd, atmwrt, rpcCall, rpcRtn* denoting reads, writes, returns, atomic reads, atomic writes, rpc calls and rpc returns respectivly. For example, $rd : [?, obs, 1]$ is a read event, where the first parameter is unspecified, the second is *obs*, which stands for an observable event, and the last parameter 1 denotes the process id. A similar notation is used for other events.

The Mona tool is currently not able to handle automata of the size corresponding to the distributed program just discussed. Hence we prove the correctness of the implementation by using our composition rule. The obvious idea is to try whether

$$\phi_{P_i}(\alpha) \Rightarrow \phi_{S_i}(\alpha)$$

holds (for $i = 1$ or $i = 2$; the formulas are symmetric). The Mona tool, however, quickly determines that this formula is not valid. There is a counter-example of length 12:

rd:[*obs*], *rpcCall, rd, atmrd, rtn, rpcRtn,*
rpcCall, rd, atmrd, rtn, rpcRtn, rtn:[*obs*],

where we have left out most of the parameters. The counter-example arises because the specification system requires exactly one atomic read in every successful read call, whereas the implementation is allowed to retry on failure.

Fortunately, we can let Mona establish

$$\phi_{P_i}(\alpha) \quad \Rightarrow \quad \exists \beta.\, \hat{\mathcal{R}}_i(\alpha, \beta), \tag{6}$$

where $\hat{\mathcal{R}}_i(\alpha, \beta) \stackrel{\text{def}}{\equiv} \phi^{Obs}_{P_i,S_i}(\alpha, \beta) \wedge \phi_{S_i}(\alpha, \beta)$ is the canonical trace abstraction. Thus, ϕ_{P_i} implements ϕ_{S_i}.

To avoid explicitly modeling the whole system at the implementation level, we use the proof rule for

compound systems. The compatibility premise of Theorem 2.2 becomes:

$$\bigwedge_i \exists \beta_i : \hat{\mathcal{R}}_i(\alpha, \beta_i) \quad \Rightarrow \quad \exists \beta : \bigwedge_i \hat{\mathcal{R}}_i(\alpha, \beta). \qquad (7)$$

However, the existential quantification on the right hand side of the implication leads to a state explosion that cannot be handled by the Mona tool.

Instead, we can exploit the information that the counter-example provided to formulate a more precise trace abstraction. So we have defined predicates that in more detail describe how internal events at one level relate to internal events at the other level. For example, we may add our intuition that between any successful read and its corresponding return at the implementation level only the last atomic read is mapped to an atomic read on the specification level. This formula, which we denote by ψ_i, looks like:

$$\begin{aligned}
&\forall t_1, t_2.\ (t_1 < t_2 \wedge \\
&\alpha(t_1) = rd : [?, obs, i] \wedge \\
&\alpha(t_2) = rtn : [?, ?, normal, obs, i] \wedge \\
&\neg\phi^{Between}_{rd:[?,obs,i]}(\alpha, t_1, t_2) \wedge \\
&\neg\phi^{Between}_{wrt:[?,?,obs,i]}(\alpha, t_1, t_2)) \\
\Rightarrow\ & \\
&(\exists t.\, t_1 < t < t_2 \wedge \\
&\alpha(t) = \beta(t) = atmrd : [?, ?, ?, i] \wedge \\
&\neg\phi^{Between}_{atmrd:[?,?,?,i]}(\alpha, t, t_2) \wedge \\
&\neg\phi^{Between}_{atmrd:[?,?,?,i]}(\beta, t_1, t) \wedge \\
&\neg\phi^{Between}_{atmrd:[?,?,?,i]}(\beta, t, t_2)).
\end{aligned}$$

We define the new trace abstractions $\mathcal{R}_i(\alpha, \beta)$ to be equal to $\hat{\mathcal{R}}_i$ conjoined with the ψ_i and two other similar predicates (one stating that any event on the program level—but an atomic read—is matched by the same event on the specification level; the other stating that an atomic read event on the program level is matched by either an atomic read event or a τ event on the specification level). With $\mathcal{R}_i$, the Mona tool proves formulas (6) and (7) within minutes.

The compatibility property (7) is stated in a single M2L formula of size 10^5 with approximately 32 visible variables at the level of deepest nesting (corresponding to an alphabet size of 2^{32}). During its processing automata with millions of BDD nodes are created. The proof required user intervention in the form of an explicit (but natural) ordering of BDD variables. Also, we have supplied a little information about evaluation order in the form of parentheses.

6 Conclusion

We have offered a practical alternative to the use of refinement mappings. We have indicated how the user contribution of information about behavioral similarities directly can be used to reduce the computational work involved in guessing internal events when two distributed systems are compared.

Our method is entirely formulated within M2L: state machines, temporal properties, finite domains, and verification rules all take on the syntax of the Mona system.

Our experiments show that very complex temporal logic formulas on finite segments of time can be decided in practice—quite in contrast to the situation for temporal logic on the natural numbers.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] M. Abadi and L. Lamport. Conjoining specifications. Technical Report Report 118, Digital Equipment Corporation, Systems Research Center, 1993.

[3] M. Broy and L. Lamport. Specification problem, 1994. A case study for the Dagstuhl Seminar 9439.

[4] E. M. Clark, I.A. Browne, and R.P Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. In A. Arnold, editor, *CAAP, LNCS 431*, pages 103–116, 1990.

[5] E. M. Clarke, O. Grumberg, and D. E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency*, pages 124–175. Springer-Verlag, 1993. Lecture Notes in

Computer Science, Vol. 803, Proceedings of the REX School/Symposium, Noordwijkerhout, The Netherlands, June 1993.

[6] U. Engberg, Grønning P., and Lamport L. Mechanical verification of concurrent systems with tla. In *Computer Aided Verification, CAV '92*. Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 663.

[7] Manna Z. et al. Step: The stanford temporal prover. In *Theory and Practice of Software Development (TAPSOFT)*. Springer-Verlag, 1995. Lecture Notes in Computer Science, Vol. 915.

[8] J.G. Henriksen, O.J.L. Jensen, M.E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A.B. Sandholm. Mona: Monadic second-order logic in practice. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, *Procedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 58–73, 1995. BRICS Notes Series NS-95-2.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[10] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification and simulation and refinement. In *A Decade of Concurrency*, pages 273–346. ACM, Springer-Verlag, 1993. Lecture Notes in Computer Science, Vol. 803, Proceedings of the REX School/Symposium, Noordwijkerhout, The Netherlands, June 1993.

[11] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in automated verification based on trace abstractions. Technical Report RS-96-?, BRICS, Aarhus University, 1996. In preparation.

[12] N. Klarlund and F.B. Schneider. Proving nondeterministically specified safety properties using progress measures. *Information and Computation*, 107(1):151–170, 1993.

[13] R. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[14] R. P. Kurshan, M. Merritt, A. Orda, and S. R. Sachs. Modelling asynchrony with a synchronous model. In *Computer Aided Verification, CAV '95, LNCS*, 1995. Lecture Notes in Computer Science.

[15] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[16] L. Lamport. The temporal logic of actions. Technical Report Report 70, Digital Equipment Corporation, Systems Research Center, 1994. To appear in Transactions on Programming Language and Systems.

[17] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. Sixth Symp. on the Principles of Distributed Computing*, pages 137–151. ACM, 1987.

[18] N. Lynch and F. W. Vaandrager. Forward and backward simulations – part i: untimed systems. Technical Report CS-R9313, Centrum voor Wiskunde en Informatica, CWI, Computer Science/Department of Software Technology, 1993.

[19] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[20] A.P. Sistla. On verifying that a concurrent program satisfies a nondeterministic specification. *Information Processing Letters*, 32(1):17–24, July 1989.