Data Independence with Generalised Predicate Symbols

Ranko Lazić* and Bill Roscoe†
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, UK

Abstract

*Where a concurrent system Impl is parameterised by a data type X with respect to which it is data independent, it is often possible to prove that Impl has a property Spec for all X by verifying it for a finite number of instantiations of X (*a threshold collection*).*

In this paper we show how to extend the definition of data independence for CSP to allow concurrent systems to include symbols representing generalised predicates*, i.e. mappings from the type X to fixed finite types such as the two-element type of booleans. Some of the main theorems that provide threshold collections, which were originally proved for data independence without these symbols, are revised for this extension.*

Keywords: *data independence, symbolic execution, predicate symbols, model checking, CSP*

1 Introduction

As the technology of model checking advances and it is used in practical system developments, the greatest obstacle is the *state explosion problem.* For this reason, a considerable amount of effort is directed towards overcoming this. One approach to this is to attempt to prevent the growth in states caused by putting state machines in parallel from being multiplied by the range of values that data variables within these components can take. Two closely related techniques for achieving this are *data independence* [14, 6, 5, 7] and exploring state spaces with symbolic rather than expanded data [4, 13, 5, 1].

For a concurrent system P to be *data independent* in type X, it must make sense for any nonempty X; thus data independence is closely related to the concept of *polymorphism* from (often functional) programming. The intention of the concept of data independence is that concurrent systems satisfying it must handle X sufficiently simply that in most cases we can expect the essential range of their behaviour not to depend on the size of X beyond some limit. Over some languages this requires some restrictions beyond polymorphism, but in [8], a combination of the process algebra CSP and a typed λ-calculus is developed in which these two conditions are identical. For this language, a wide range of theorems are proved about data independence and its consequences, using both operational semantics with symbolic data and the theory of *logical relations* [10, 11].

One of the most important factors in allowing proofs of data independence theorems is the way the conditions imposed restrict the ways in which the control flow of programs can depend on values of the type X. In the language of [8], the only way permitted is equality tests between values, and even permitting these has a significant impact on the results obtained: typically, without equality checks, we can limit the type size to no more than 2, while with them we get often much larger calculated *threshold collections.* It was therefore natural that our original definition of data independence forbade the use of functions from X to other types, as the use of such functions (imagine, for example, an

*Also attached to the Mathematical Institute, Belgrade University. Supported by a Junior Research Fellowship at Christ Church (Oxford), and previously by a Domus and Harmsworth Senior Scholarship at Merton College (Oxford) and by a scholarship from Hajrija & Boris Vukobrat and Copechim France SA.

†Supported by grants from the US ONR.

injective function from X to the natural numbers) followed by unrestrained use of the result type could allow just about all restrictions on X to be bypassed.

If, however, we restrict the functions on X to be to fixed (i.e., not changing as X does) finite types, then the situation is different. In essence, such a function (necessarily a formal function symbol rather than a concrete function, as the latter would fix X) simply represents a partition of X into finitely many subtypes: the inverse images of each member of the range type U. It is reasonable to hope that we might find that we can get threshold collections for these subtypes, and therefore find ways of proving a property, independent both of X and the interpretation of the function symbol, using a finite number of checks. The purpose of this paper is to show that this is so, extending both the definition of data independence and some of the main theorems of [8] to accommodate function symbols of this sort.

In many applications of this work, the function symbols are *predicate* symbols, i.e. the range type U is the type of booleans. These can be used to represent (and thereby to prove results about) an arbitrary subset of X. Indeed, because of the importance of this case, and to emphasise that U is fixed and finite in all cases, we will henceforth refer to the function symbols as *generalised* (where appropriate) *predicate symbols*. We can, however, use this extension to deal with any application where we know X is to be divided into a fixed number of pieces. In [9], we showed how proofs of the property of determinism by data independence could be extended to encompass these generalised predicate symbols, which in the case study in that paper were mappings from the type of record names to a fixed set of security levels.[1]

As can be seen in [8], the theory of data independence in CSP is complex and relies on detailed arguments. Space constraints mean that all we can do in this paper is state the main principles governing the extension, and state our theorems together with a brief indication of how the earlier proofs are modified.

Detailed comparisons of our previous work on data independence with work by other authors can be found in [8, 7]. The most related to the extension by generalised predicate symbols seems to be [5], where predicates of the forms $(x \bmod m) = d$ and $(x \bmod m) < d$ on natural numbers x are considered.

2 The language

The language we use for expressing concurrent systems is a combination of the process algebra CSP [12] and a typed λ-calculus [10]. The combination is such that, in addition to type constructs coming from the λ-calculus, the language has *process types*. Concurrent systems are then terms of process types, and CSP operators can be used to build them. As in CSP, there is no conceptual difference between concurrent systems and their component processes, so refer to both as 'processes'.

It may appear that this is an unnecessarily heavy choice of formalism. However, it is strongly motivated by practical model checking, as this language is a smaller and more theoretically presented version of CSP_M [12], which is the version of CSP used in tools. The typed λ-calculus is needed for computing data which the processes communicate across channels, and for manipulating networks of processes. It also combines well with CSP because both formalisms are declarative.

Due to space limitations, in this paper we can only present the syntax of types, and remark briefly about the syntax of terms. The same language is used in [8], to which the reader is referred for details.

We assume there is a set *AllTypeVars* of all type variables, and that there are subsets of it *NEFTypeVars* and *FinTypeVars* such that each of

$NEFTypeVars \cap FinTypeVars$,
$NEFTypeVars \setminus FinTypeVars$,

[1] Determinism, which has close links to security, is a more specialised property than the ones covered in the present paper, and requires somewhat different styles of argument to verify. Thus we are showing how an extension of data independence originally shown to be valid only in the special case also applies in the mainstream.

$FinTypeVars \setminus NEFTypeVars$,
$AllTypeVars \setminus (NEFTypeVars \cup FinTypeVars)$

is countably infinite.

A type variable $X \in AllTypeVars$ can be substituted by any type T, except that if $X \in NEFTypeVars$ then all denotational interpretations of T (see Section 3) must be nonempty sets with the flat ordering, and if $X \in FinTypeVars$ then they must be partial orders whose sets are finite.

The types of the language are defined by

$$\begin{array}{rcl} T & ::= & X \mid id_1.T_1 + \cdots + id_n.T_n \mid \\ & & T_1 \times \cdots \times T_n \mid \mu\, X.T \mid \\ & & T_1 \to T_2 \mid Proc_{T,\phi} \end{array}$$

where:

- X ranges over $AllTypeVars$.
- In each sum type, $id_1, \ldots, id_n$ are some mutually distinct identifiers, which are used to separate the component types.
- In each inductive type $\mu\, X.T$, X must be neither from $NEFTypeVars$ nor from $FinTypeVars$, and any free occurrence of X in T must be only within sums, products and μs. Those free occurrences become bound by the outermost μ.
- In each process type $Proc_{T,\phi}$, T must be of the form $\sum_{i=1}^{n} id_i.\prod_{j=1}^{m_i} T_{i,j}$ where $n \geq 1$ and any $T_{i,j}$ is such that all its denotational interpretations (see Section 3) are nonempty sets with the flat ordering and such that it does not involve function types or process types. Processes of this type may only perform visible events (i.e. visible actions) which are elements of the type T, so that T is thought of as the alphabet, and each identifier id_i is thought of as a channel of type $\prod_{j=1}^{m_i} T_{i,j}$.

  The second index ϕ specifies the ways in which the components of the channels in T may be used. For any i and j, we have that $\phi(id_i, j) \subseteq \{!, ?, \$\}$, where the three symbols represent outputs, inputs and nondeterministic selections.

Example 2.1 The type constructs that we have can be used to define many commonly used types, such as the type of booleans, the type of all finite lists of elements of a type V, and the type of all natural numbers:

$$\begin{array}{l} Bool = true.1 + false.1 \\ List(V) = \mu\, X.(nil.1 + cons.(V \times X)) \\ Num = \mu\, Y.(zero.1 + succ.Y) \end{array}$$

where the singleton type 1 is a product type with no components, and where X does not occur free in V. ■

The terms of the language are those that appear in type judgements which are derivable from the typing rules given in [8, Section 2.5]. A type judgement is of the form

$$\Gamma \vdash r : T$$

where r is a term, Γ is a type context which is of the form $\{x_1 : T_1, \ldots, x_k : T_k\}$ and which provides types for term variables that occur free in r, and T is a type of r. The phrase 'a term r such that $\Gamma \vdash r : T$ is derivable' will be abbreviated by 'a term $\Gamma \vdash r : T$'.

The λ-calculus term constructs are those for introducing and eliminating elements of sum types, product types, inductive types and function types, and a construct for equality tests. The CSP term constructs are those for $STOP$, prefixing (including outputs, inputs and nondeterministic selections), binary external and nondeterministic choices, external and nondeterministic choices replicated over certain types, parallel composition, hiding and recursion.

3 Denotational semantics

If T is a type, we write

$$[\![T]\!]^M_\delta$$

for the partial order which is the denotational interpretation of T with respect to M and δ, where:

- M is one of Tr, Fa, FD, which represent the finite-traces, stable-failures and failures/divergences models of CSP. It specifies which of those models is used for interpreting process types within T.

- δ assigns a partial order to each type variable $X \in \mathit{AllTypeVars}$, where any $X \in \mathit{NEFTypeVars}$ may only be assigned nonempty sets with the flat ordering, and any $X \in \mathit{FinTypeVars}$ may only be assigned partial orders whose sets are finite.

If $\Gamma \vdash r : T$ is a term, we write

$$[\![\Gamma \vdash r]\!]^M_{\delta,\eta}$$

for its denotational interpretation (which does not depend on the type T) with respect to M, δ and η, where:

- M and δ are as above.
- η is an assignment to Γ with respect to M and δ, or more precisely $\eta[\![x_i]\!] \in [\![T_i]\!]^M_\delta$ for each $(x_i : T_i) \in \Gamma$.

The definition of the denotational semantics, and a sketch of how well-definedness is proved, can be found in [8, Sections 2.4 and 2.5].

For technical reasons, whenever one works with $M = \mathrm{FD}$, one may consider only finitely nondeterministic CSP constructs and only process types $\mathit{Proc}_{T,\phi}$ such that any type variable X which occurs free in T is from $\mathit{FinTypeVars}$.

As a notational convention, we may omit one or more of M, δ and η in cases in which the denotational interpretation does not depend on them.

4 Refinement

The ordering which is usually associated with the three models of CSP mentioned in Section 3 is the reverse subset ordering. Thus, in the finite-traces model, a process is 'greater' than another if it has fewer traces. In the other two models, the ordering is pointwise, so that for instance both fewer failures and fewer divergences are required.

This ordering is known as the *refinement* ordering, because when a process has fewer behaviours than another, one can think of it as less nondeterministic, and thus 'better' from the point of view of any user.

For CSP, and consequently for the language in this paper, refinement provides the notion of correctness of processes: both specifications (i.e. correctness conditions) and implementations are processes, and the specification is satisfied if its is refined by the implementation. Indeed, this is the notion of correctness in the theorems in Section 6.

Refinement is adequate as a notion of correctness because of the wealth of CSP operators, such as nondeterministic choice and hiding, which make it possible to write correctness conditions as processes and to extract relevant aspects of implementations. Secondly, no extra work is required to support methods such as compositional verification and step-wise verification. For more on this topic, see [12, particularly pp. 46–47].

Definition 4.1 If $\Gamma \vdash \mathit{Spec} : \mathit{Proc}_{T,\phi}$ and $\Gamma \vdash \mathit{Impl} : \mathit{Proc}_{T,\psi}$ are two processes (i.e. terms of process types) with a common alphabet T,

$$\mathit{Spec} \sqsubseteq^M_{\delta,\eta} \mathit{Impl}$$

will mean that Spec is refined by Impl with respect to M, δ and η, where Γ will always be clear from the context, and M, δ and η are as in Section 3. This is the case if and only if

$$[\![\Gamma \vdash \mathit{Spec}]\!]^M_{\delta,\eta} \supseteq [\![\Gamma \vdash \mathit{Impl}]\!]^M_{\delta,\eta}$$

where if $M = \mathrm{Fa}$ or $M = \mathrm{FD}$, $\supseteq$ represents pointwise reverse subset. ■

5 Data independence with a generalised predicate symbol

When a process is data independent in X with a generalised predicate symbol d, it should be possible to vary X, and d should be a function symbol from X into a type U which is fixed (i.e. with no free type variables), nonempty and finite. In the language in this paper, these correspond to X being a type variable, and d being a term variable of type $X \to U$.

For simplicity, X is assumed to be from $\mathit{NEFTypeVars}$, and U is assumed not to involve function types. Also, we concentrate on the case when the process does not have any free type variables other than X. When there are more than one free type variable, they can for example be analysed manually one at a time by substituting fixed data types for the rest.

Definition 5.1 A process $\Gamma \vdash P : Proc_{T,\phi}$ is said to be data independent in X with a generalised predicate symbol d if and only if:

- X is from *NEFTypeVars*.
- $\Gamma \vdash P : Proc_{T,\phi}$ has no free type variables other than X.
- $\Gamma = \{d : X \to U\}$ for some type U built only from sums with at least one component and products, where the latter may have no components, in which case they are singleton types. (The denotational interpretation $[\![U]\!]$ is then a nonempty finite set with the flat ordering.) ■

With various degrees of frequency, each of the following kinds of processes either are data independent or can be analysed using data independence: communication protocols, in the type of values they communicate; memories, databases and systems involving them, in the types of addresses and values; agents and the spy in security protocols, in the types of agent identities, nonces and keys; nodes in a system consisting of many nodes, in the type of their identities.

The need for generalised predicate symbols arises when a process is data independent in X except that it uses a partitioning of X into a finite number of subtypes, since this is equivalent to a generalised predicate symbol such that the subtypes are the inverse images of the elements of its range. A range of examples which is a major motivation for this paper comes from the recent work on combining induction and data independence in order to verify systems with arbitrary network topologies [2, 3], where isolating one subtype of X (which can be done with a predicate symbol, whose range is the type *Bool*) is required.

6 Threshold collections

We can now state two theorems which provide threshold collections. Given processes *Spec* and *Impl* which are data independent in X with a generalised predicate symbol d, these are finite collections of interpretations of X and d such that if *Spec* is refined by *Impl* for all those interpretations, then the refinement holds for all interpretations, or for all interpretations with an appropriate property.

For simplicity, we assume that the processes *Impl* do not use nondeterministic selections.

(The theorems were obtained by extending simplified versions of Theorems 5.3.6 and 5.4.7 in [8]. For a precise statement of the assumption about nondeterministic selections, see the beginning of Chapter 5 in [8]. Precise statements of the conditions ($\mathbf{NoEqT}_X$), (**Norm**) and ($\mathbf{RecCho}_X$), which are used in the theorems, can be found in Section 3.5 of [8].)

6.1 Theorems

The first theorem assumes that *Spec* and *Impl* satisfy a condition ($\mathbf{NoEqT}_X$) which amounts to banning any equality tests between values of type X, and that *Spec* satisfies a condition (**Norm**) which is based on banning any nondeterminism whose effects are not immediately apparent, and it states that, for any of the three models of CSP we are considering (see Section 3), if *Spec* is refined by *Impl* whenever exactly one of the inverse images of elements of U has size 2 and the rest have size 1, then the refinement holds always, where the type U is the range of the generalised predicate symbol d. It thus provides a threshold collection whose size is the size of U.

Theorem 6.1 *Suppose that:*

(i) $\{d : X \to U\} \vdash Spec : Proc_{T,\phi}$ *and* $\{d : X \to U\} \vdash Impl : Proc_{T,\psi}$ *are data independent in* X *with a generalised predicate symbol* d*;*

(ii) $\{d : X \to U\} \vdash Spec : Proc_{T,\phi}$ *and* $\{d : X \to U\} \vdash Impl : Proc_{T,\psi}$ *satisfy* ($\mathbf{NoEqT}_X$)*;*

(iii) $\{d : X \to U\} \vdash Spec : Proc_{T,\phi}$ *satisfies* (**Norm**)*.*

For any $a \in [\![U]\!]$, *let*

$$\begin{array}{l}\delta_a[\![X]\!] = \{(1,a') \mid a' \in [\![U]\!]\} \cup \{(2,a)\} \\ (\eta_a[\![d]\!])((1,a')) = a' \\ (\eta_a[\![d]\!])((2,a)) = a\end{array}$$

For any $M \in \{\mathrm{Tr}, \mathrm{Fa}, \mathrm{FD}\}$, *if*

$$Spec \sqsubseteq^M_{\delta_a,\eta_a} Impl$$

for all $a \in [\![U]\!]$, *then*

$$Spec \sqsubseteq^M_{\delta',\eta'} Impl$$

for all δ' *and* η'. ■

The second theorem assumes that *Spec* satisfies a condition (**RecCho**$_X$) which is much weaker than (**Norm**) and amounts to banning a certain kind of nondeterministic choice of values of type X, and it states that if *Spec* is refined by *Impl* when each of the inverse images of elements of U has size ν, then the refinement holds either always (for the finite-traces model) or whenever the size of the interpretation of X is at least 2 (for the other two models).

The quantity ν is computed as the sum of the following three quantities: the supremum of the number of distinct values of type X that a normalised version of *Spec* (see Section 6.2) ever has to store for future use, the supremum of the number of such values that *Impl* ever has to store for future use, and the supremum of the number of such values that can ever be either input or nondeterministically chosen in a single event of *Impl*. (For precise definitions, in a context without generalised predicate symbols, see Definitions 3.3.3 and 3.7.2 in [8].)

If either $\nu = 0$, or $M = \mathrm{FD}$ and ν is infinite, the theorem does not apply because of the restrictions on interpretations of X (see Definition 5.1, and the assumption stated near the end of Section 3).

Theorem 6.2 *Suppose that:*

(i) $\{d : X \to U\} \vdash Spec : Proc_{T,\phi}$ *and* $\{d : X \to U\} \vdash Impl : Proc_{T,\psi}$ *are data independent in* X *with a generalised predicate symbol* d*;*

(ii) $\{d : X \to U\} \vdash Spec : Proc_{T,\phi}$ *satisfies* (**RecCho**$_X$).

Let

$$\begin{array}{rcl}\nu & = & W_{\mathcal{N}}(\{d : X \to U\} \vdash Spec) + \\ & & W(\{d : X \to U\} \vdash Impl) + \\ & & \max\{L_?(\{d : X \to U\} \vdash Impl), \\ & & L_\sqcap(\{d : X \to U\} \vdash Impl)\}\end{array}$$

and let

$$\begin{array}{l}\delta[\![X]\!] = [\![U]\!] \times \{k \in \mathbb{N} \mid k < \nu\} \\ (\eta[\![d]\!])((a',k)) = a'\end{array}$$

For any $M \in \{\mathrm{Tr}, \mathrm{Fa}, \mathrm{FD}\}$, *if*

$$Spec \sqsubseteq^M_{\delta,\eta} Impl$$

then

$$Spec \sqsubseteq^M_{\delta',\eta'} Impl$$

for all δ' *and* η' *such that* $|\delta'[\![X]\!]| \geq 2$.

For $M = \mathrm{Tr}$, *the conclusion is also valid for* $|\delta'[\![X]\!]| = 1$. ■

6.2 About proofs

As we have remarked, the two theorems we have presented are based on Theorems 5.3.6 and 5.4.7 in [8]. The four main components of the proofs of those theorems (see [8]) are: symbolic operational semantics, in which variables of type X are left as symbols; symbolic normalisation, a procedure for transforming symbolic labelled transition systems into a form where any trace (i.e. sequence of visible actions) can be executed in at most one way; logical relations, which provide an entirely denotational way of reasoning about data independence; eliminating symmetry, where two states of a refinement check are identified if and only if they can be obtained one from another by a permutation of the interpretation of X.

The two theorems we have presented can be proved along the same lines, with two significant differences. The first is that symbolic labelled transition systems are extended by branchings which ensure that, at any node, the result of applying the generalised predicate symbol d to any variable is known. The second is that, in eliminating symmetry, only permutations which preserve the interpretation of d are considered. Logical relations do not need to be revised, as they were not restricted to (weakly) data independent processes.

6.3 Beyond a single generalised predicate symbol and no constant symbols

For simplicity, we have been considering a single generalised predicate symbol and no constant symbols.

In a theoretical sense, adding finitely many more generalised predicate symbols actually has no effect since, given symbols $d_1, \ldots, d_k$ whose ranges are $U_1, \ldots, U_k$, we can replace them by a single one $\hat{d}$ whose range is $U_1 \times \cdots \times U_k$, with each use $d_i(\cdot)$ being substituted by $pr_i(\hat{d}(\cdot))$.

Given the statements of our theorems, this multiplication of ranges has serious effects on the size of the threshold collection in Theorem 6.1 and on the sizes of the interpretations of X in the threshold collections in both theorems. However, this complexity can be reduced by if we can assume (presumably, on the basis of something we know about the application) some relationships among the images of each element of X under $d_1, \ldots, d_k$.

A finite number of constant symbols (or 'constants') of type X can also be allowed. Threshold collections can then be partitioned, one subcollection for each matrix of results of applying the generalised predicate symbols to the constants (or for each such matrix which makes sense for the application). Consequently, in the statements of the theorems, we can focus on only one such matrix, and then the constants can be incorporated very similarly to how that was done in [8].

References

[1] Corella, F., Z. Zhou, X. Song, M. Langevin and E. Cerny, *Multiway Decision Graphs for Automated Hardware Verification*, Formal Methods in System Design 10 (1), 7–46, Kluwer, 1997.

[2] Creese, S.J. and A.W. Roscoe, *Verifying an Infinite Family of Inductions Simultaneously Using Data Independence and FDR*, submitted to FORTE/PSTV, 1999.
Also at: `http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/textmat/`

[3] Creese, S.J. and A.W. Roscoe, *Formal Verification of Arbitrary Network Topologies*, in the same volume as this paper.

[4] Hennessy, M. and H. Lin, *Symbolic Bisimulations*, Theoretical Computer Science 138 (2), 353–389, 1995.

[5] Hojati, R. and R.K. Brayton, *Automatic Datapath Abstraction in Hardware Systems*, Proceedings of the 7th CAV, Springer LNCS 939, 98–113, 1995.

[6] Ip, C.N. and D.L. Dill, *Better Verification Through Symmetry*, Formal Methods in System Design 9 (1–2) (Special Issue on Symmetry in Automatic Verification), 41–75, Kluwer, 1996.

[7] Lazić, R.S., *Some Literature on Data Independence and Parameterised Verification*, an informal literature survey, 1998.
Available at: `http://www.formal.demon.co.uk/workshop/199803`

[8] Lazić, R.S., *A Semantic Study of Data Independence with Applications to Model Checking*, D.Phil. thesis, Oxford University Computing Laboratory, to appear in 1999.
A complete draft is available at: `http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/textmat/`

[9] Lazić, R.S. and A.W. Roscoe, *Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays*, Informal proceedings of the 3rd INFINITY (1998), to appear as a Research Report of the Technical University of Munich.
A fuller version is the Technical Report PRG-TR-2-98, Oxford University Computing Laboratory, `http://www.comlab.ox.ac.uk/oucl/publications/tr.html`

[10] Mitchell, J.C., *Type Systems for Programming Languages*, Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, 365–458, Elsevier, 1990.

[11] O'Hearn, P., *Parametric Polymorphism*, in 'Domains and Denotational Semantics: History, Accomplishments and Open Problems', Bulletin of the EATCS 59, 227–256, 1996.

[12] Roscoe, A.W., *The Theory and Practice of Concurrency*, Prentice Hall, 1998.
The associated web site is: `http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/`

[13] Seger, C.-J.H. and R.E. Bryant, *Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories*, Formal Methods in System Design 6 (2), 147–190, Kluwer, 1995.

[14] Wolper, P., *Expressing Interesting Properties of Programs in Propositional Temporal Logic*, Proceedings of the 13th ACM POPL, 184–193, 1986.