# Relating Data Independent Trace Checks in CSP with UNITY Reachability under a Normality Assumption

Xu Wang[1], A.W. Roscoe[1], and R.S. Lazić[2]

[1] Oxford University Computing Laboratory,
Oxford OX1 3QD, UK. {xu.wang,bill.roscoe}@comlab.ox.ac.uk
[2] Department of Computer Science, University of Warwick,
Coventry CV4 7AL, UK. ranko.lazic@dcs.warwick.ac.uk

**Abstract.** This paper shows how to translate the problem of deciding trace refinement between two data independent (DI) CSP processes to an unreachability problem in a DI Unity program. We cover here the straightforward but practically useful case when the specification satisfies a normality condition, **Norm**, meaning that we do not have to worry about hidden or *unrecorded*[1] data variables. This allows us to transfer results about the decidability of verification problems involving programs with data independent arrays from UNITY to CSP.

**Keywords:** CSP, Data independence, Trace refinement, Model checking, Reachability, Array.

## 1 Introduction

Algorithmic formal verification of hardware and software systems is an important way of establishing system correctness or to debugging (whichever case applies!). *State exploration* of various types is commonly employed. Whereas state exploration of control structures is usually effective and efficient, handling data is more challenging. Often the only way of including data into correctness checks is to branch it into control structure, so that two instances of a control state with different data become, in effect, different control states. This often creates enormous state explosions and thus limits the techniques' applicability. We seek to improve our understanding of the special properties of data in algorithmic verification to reduce this problem.

The problem of algorithmic verification of data-bearing systems in general is hard. But an important subclass of data-bearing systems where useful results are available is the *data independent* ones. Intuitively, data independence means that data is opaque: although it can be communicated (input and output) and stored, the only way to probe the information inside is by equality testing with

[1] Roughly speaking, unrecorded variables are DI variables assuming values internally generated (e.g., by DI replicated choice as in Section 4.1), rather than inputted from environment.

another of its kind. Thus a data independent type $X$ is nothing more than an abstract "set". As sets of equal size are substitutable via bijection, only the size of $X$ matters.[2]

Typical examples of data independence are: types of data stored in buffers or transmitted by protocols, and types of addresses/pointers in a cache or memory (equality testing is used to check if two pointers point to the same location).

Data independence has been studied in the past from the process algebra [2] and temporal logic [3] perspectives. Our research has explored both of these [4, 5] and we have formulated a unifying semantic framework to capture data independence [6,7].

While some past work on data independence assumes the infinity of DI types[2,3], no such requirement is present in our work. Actually, our work can handle infinite families of parameterised systems, where the type $X$ can be of arbitrary nonempty finite size or infinite. The verification problem we face is more than just the uniform verification of parameterised systems [8].

Data independence is also related to another important technique in analysing data-bearing systems: symbolic labelled transition systems [9,10]. A symbolic LTS tries to represent, in a finite graph, a system which is potentially infinite thanks to data-bearing. Variables are left as symbols instead of being instantiated by concrete values. Symbols and their properties (described in a boolean expression) are used in the system so that a set of concrete value instantiations is treated in one go instead of each concrete value instantiation in a separate go.

Other forms of symbolism (e.g., those based on binary decision diagram, regular language, regions, etc.) also flourish in model/refinement checking for overcoming problems like the tractability of large state spaces, abstraction of unbounded structures (like queues, stacks, linear integer, and parameterised linear topology), and abstraction of real time and hybrid systems. The relationship between these works and data independence has yet to be fully explored.

The overall aim of the present paper is to relate the DI work that we have done more recently on a Unity-like language to that we did previously in CSP. Besides the obvious benefits of connecting and unifying different data independence theories, there are some other very important practical ones, since it means that results obtained in the Unity style may now apply to CSP. One particular class of result we want to borrow are our recent decidability results on DI arrays[11,5,12].

Secondly, in practical applications, both approaches have relative merits and drawbacks. The benefits of our[3] process algebraic framework are: unified language, compositionality, and stepwise refinement, while the benefits of a temporal logic approach are: abstract property and liveness/fairness friendliness. So it would often be very instructive if some light could be shed on how the same case study could be differently encoded and processed in different approaches.

[2] It is sometimes possible to weaken the definition of data independence in carefully controlled ways, such as allowing symbols representing functions from the DI type to a known finite type such as the booleans[1].

[3] Some of these are specific to CSP.

And the most direct way to show that is by a translation between the checking problems of the two frameworks.

Thus, the rest of this paper will aim to give a two-step translation procedure that can, through an intermediate SLTS, automatically convert a CSP DI refinement checking problem to an equivalent Unity DI model checking problem, namely unreachability.

$$\mathit{Spec} \sqsubseteq_X \mathit{Impl} \quad \overset{\mathit{translate}}{\Longrightarrow} \quad \mathit{Prog} \models_X \mathit{Unreachability}$$

The reasons why we restrict the property to reachability/unreachability are

– Reachability is a powerful concept. Both trace refinement and (we believe) stable-failure refinement can be encoded into it. However it cannot be true for failure-divergence refinement because of the infinitary nature of divergence: an effective translation would imply solubility of the halting problem.
– By restricting ourselves to reachability, decidability results on more powerful classes of programs with DI arrays [11] are achievable. This extension to our reasoning power with arrays contains a large proportion of likely practical applications.
– Reachability is a natural target for our translation problem, and corresponds to the intuition that both trace properties and unreachability correspond naturally to *safety* specifications.

## 2 The Norm Condition and This Paper

Translating from the CSP language to a Unity fragment language is not easy, especially since there is a mismatch in the expressive power of the two languages.

The Unity fragment used in [11,5] is a much simpler language than CSP. In essence, its programs consist of a finite set of variables, an initialisation of the variables, and a finite set of guarded multiple assignments of the form, $\mathit{boolean\_expression} \longrightarrow \{\mathit{assignment}\}$.

Simplicity buys unreachability checking on Unity DI programs a useful property: monotonicity.

$$\mathit{Prog} \models_{T_1} \mathit{Unreachability} \wedge |T_2| \leq |T_1| \Rightarrow \mathit{Prog} \models_{T_2} \mathit{Unreachability}$$

Similar properties, however, are not enjoyed by CSP refinement checking in any of its three major models. In CSP we even can construct $(\mathit{Spec}, \mathit{Impl})$ pair that refines (in all three models) iff the size of $T$ is an odd number (c.f. Chapter 5 in [4]). So some $\mathit{Spec} \sqsubseteq \mathit{Impl}$ are inherently untranslatable to $\mathit{Prog} \models \mathit{Unreachability}$.

Another symptom of the language power mismatch is found in the handling of arrays.

*Some CSP processes can use DI variables to simulate a limited form of DI arrays.*

A good example of that is the specification of a nondeterministic register.

$$\begin{array}{rl} \mathit{Spec}_{NR} \equiv & \mathsf{let}\ NR(x) = out!x \to NR(x) \\ & \qquad\qquad \Box \\ & \qquad\qquad in?y_0 \to\ NR(x) \sqcap NR(y_0) \\ & \mathsf{within}\ in?x_0 \to NR(x_0) \end{array}$$

When considering the effect of this process as the left-hand side of a refinement check, the parameter variable $x$ above actually encodes a boolean DI array (i.e., a DI set); a more intuitive equivalent representation using explicit array variables is[4],

$$\begin{array}{rl} \mathit{Spec}'_{NR} \equiv & \mathsf{let}\ NR(arr) = out\$x : arr[x] \to NR(\{x\}) \\ & \qquad\qquad \Box \\ & \qquad\qquad in?y_0 \to\ NR(arr \cup \{y_0\}) \\ & \mathsf{within}\ in?x_0 \to NR(\{x_0\}) \end{array}$$

The root of the problem with $\mathit{Spec}_{NR}$ is that many nondeterministic choices may be made before their effects are shown by output. In $\mathit{Spec}'_{NR}$ above, in order to delay the choice until its effect is seen, we need to introduce an explicit array. So $\mathit{Spec}_{NR}$ effectively enjoys the power of arrays even without introducing them.[5]

But this cannot be true in Unity, where it can be shown to be impossible to simulate DI arrays using simple DI variables. Actually, to translate $\mathit{Spec}_{NR} \sqsubseteq_X \mathit{Impl}_{NR}$ to Unity for some $\mathit{Impl}_{NR}$, $\mathit{Spec}_{NR}$ need be first transformed to something like $\mathit{Spec}'_{NR}$ in order to bring out the hidden arrays in it. This is called *normalisation*; that is, transforming a specification to a form satisfying *the* **Norm** *condition* below.

**Norm**
A process satisfies this if and only if

(i) *there is no replicated choice over DI types (except for input ? and nondeterministic selection \$);*
(ii) *no hiding or renaming;*
(iii) *only alphabetised parallel;*
(iv) *each branch of a multi-path choice*[6] *is disjoint with the others in the set of channels it will use for the next communication.*

**Norm** is essentially what appears in [13], which extends the definition in [4] by allowing alphabetised parallel.

**Norm** and normalisation are closely related to the traditional notion of determinisation, the algebraic normal form of CSP in Chapter 11 of [13], and the

[4] The syntax used for the example is mainly taken from $CSP_{di-SPEC}$ in Section 4.1. The only addition is the use of selective nondeterministic selection ($out\$x : arr[x]$), which is a dual of selective input (i.e., $in?x : b$) in $CSP_{di-SPEC}$.

[5] We note that this implies that in order to translate refinement questions like that of $\mathit{Spec}_{NR}$ to Unity, it is likely that arrays would be required in the Unity even if not in the CSP.

[6] Multi-path choices are the choices of form $\sqcap\{P_1, ..., P_n\}$ and $\Box\{P_1, ..., P_n\}$; binary choices are their special cases.

normal form computed by FDR [14]. To understand it fully, some explanation of nondeterminism in CSP is in order.

Nondeterminism in CSP usually means extensional nondeterminism[7], which is defined in the extensional semantics of processes. The semantics must be fine-grained enough though, like the stable failures and failures divergences models, to be able to record that nondeterminism. It means nondeterminism in externally observable behaviour, rather than nondeterminism in the graph structure of transition systems (i.e., the nondeterminism from invisible actions and multiple actions with the same labels). We call this second sort *graph nondeterminism.*

In checking CSP processes, tools do not usually calculate their extensional semantics directly. Instead, various kinds of transition graphs are used, like plain LTSs, symbolic LTSs, or GLTSs (i.e., LTSs annotated with minimal acceptances and divergence [15]). These graphs, of course, often exhibit graph nondeterminism, and it is important that we understand this and how it relates to the extensional variety.

Extensional nondeterminism usually implies graph nondeterminism; for instance, an extensionally nondeterministic process must have a graph-nondeterministic plain LTS. But it is not absolutely so for other graphs, since the same process may have a graph-deterministic GLTS.

For the purpose of this paper, we only need to consider graph nondeterminism, since the trace model is too coarse-grained to record any extensional nondeterminism. In the rest of this paper, whenever nondeterminism, determinism, or determinisation is mentioned, it means the graphical sort. Moreover, sometimes the special CSP terminology for graph determinism, like **Norm** and normalisation[8], are also used.

Many of these ideas are illustrated well by the $Spec_{NR}$ and $Spec'_{NR}$ examples above. The first gets its nondeterminism from the branching (implemented using internal actions) of the choice operator $\sqcap$. The second, extensionally equivalent process, gets its from the selection \$. The first fails **Norm** because it has the same channel on either side of the $\sqcap$, whereas the second passes it. It would be possible to eliminate all graph nondeterminism from the second by suitable labelling of the nodes where \$ appears; this is not possible for the first. There is a clear sense in which $Spec'_{NR}$ is the normal form of $Spec_{N}R$.

Going back to the language mismatch discussion above, it is now clear that normalisation and monotonicity above are the two major barriers to translating CSP refinement checks to Unity. As a matter of fact, in [16] most of the attention has been devoted to the study of the two problems. It is shown that monotonicity and normalisation are inherently related; any CSP specification that is normalisable in a defined sense will automatically possess a monotonicity property in its refinement by any implementation.

[7] For example, when we say *Spec* is more nondeterministic than *Impl*, we mean it extensionally.

[8] Strictly speaking, graph-deterministic graphs only correspond to pre-normal form graphs as they may contain semantically equivalent nodes and so not be complete normal forms yet [13].

We therefore think it is important to set out the results for this case, all the more so because most natural specification processes either satisfy **Norm** already or can be altered easily to do so. By assuming that all specifications do satisfy **Norm**, this paper shows that the translation procedure (esp., that of specifications) can be presented in a much simpler way than in [16].

The paper is organised as follows. Section 3 illustrates the general idea of the translation, while Section 4 introduces the basic formalisms and definitions used. Section 5 develops details of the translation, where Section 5.1 translates *Impl* to a Unity *generator*, Section 5.2 translates *Spec* to a Unity *acceptor*, and section 5.3 connects the two to form the whole *Prog* for unreachability checking. Section 6 concludes the paper with pointers to some promising future work.

## 3 The General Ideas

### 3.1 Generators and Acceptors

The key point of the translation from CSP refinement to Unity unreachability lies in the construction of a Unity program, *Prog*, and the identification of the states in *Prog* that are required to be unreachable.

Our initial idea about a possible solution came from the refinement checking procedure used in FDR. Later, as only the trace model is treated as a first step, we realised that it can be simplified and presented in an automata-theoretical approach following [17], which is what we now present.

The idea is to construct *Prog* to implement refinement checking on $(\mathit{Spec}, \mathit{Impl})$ by exploiting nondeterminism in Unity: the refinement will fail if and only if *Prog* will, for some set of nondeterministic choices, reach a designated control state.

– *Impl* is implemented in Unity as a nondeterministic behaviour generator, *Gen*, which generates through nondeterminism all possible traces of *Impl* and only them. (Note that the prefix-closedness of traces means all states of the automaton are accepting.)
– *Spec* is implemented in Unity as a deterministic behaviour acceptor, *Acp*, which accepts exactly the traces of *Spec*. The responsibility of an acceptor is to identify errors in the behaviour of generators.
  Basically, the idea of acceptor, like the ideas of tester process [18] and monitor automaton [19], is just another reformulation of a well-known algorithm in deciding language containment between two finite automata. The algorithm consists of two steps:
  • The first step is to determinise *Spec* and calculate its complement, $\overline{\mathit{Spec}}$.
  • The second step is to check the emptyness of the intersected language, $\overline{\mathit{Spec}} \cap \mathit{Impl}$

  Normalisation and the translation to an acceptor correspond to the first step, while unreachability checking corresponds to the second. Normalisation makes sure that all acceptors will be deterministic; that is, it never refuses a correct behaviour, and always refuses any erroneous behaviour.

– Run *Gen* and *Acp* in parallel. *Gen* outputs an event and waits. *Acp* inputs it and sees if it is acceptable. If accepted, *Acp* signals *Gen* to continue. Otherwise, an *error* occurs. (Note the two automata perform the check interactively, rather than on a word-by-word basis, as they do in Section 2.5 of [20]).

```
 |-------cont-----------|
 |                      |
 V                      |
Gen--test c.x1.x2..--->Acp--->error
```

Thus $Prog = Gen \parallel Acp$.
– Model checking to verify unreachability of *error* on *Prog*

$$Spec \sqsubseteq_X Impl \quad \overset{translate}{\Longrightarrow} \quad Prog \models_X Unreachability(error)$$

### 3.2 The Procedure

Unity is a state-based language; states are defined by value assignments on a finite set of variables. In its pure form, even control states are encoded into variables. The semantics of Unity programs (esp., DI programs) can be naturally captured by a SSTS (Symbolic State Transition System).

On the other hand, CSP is an action-based formalism. Its classic operational semantics is based on plain labelled transition systems; though for data-bearing CSP, symbolic LTSs may be more convenient from the point of view of algorithmic verification.

But the difference between SSTSs and SLTSs is not great. They can easily be translated to-and-fro just as can plain STSs and LTSs [18]. In the rest of this paper, therefore, only SLTSs will be used explicitly. Based on SLTSs, our translation procedure for implementations and specifications can be formulated as below.

$$\begin{array}{lllll} Impl & \Longrightarrow & SLTS & \Longrightarrow & Gen \\ NSpec & \Longrightarrow & SLTS_N & \Longrightarrow & Acp \end{array}$$

*NSpec* is a specification satisfying **Norm**, while $SLTS_N$ is a deterministic SLTS. *Acp* not only accepts the behaviours defined in $SLTS_N$ but also monitors the behaviours outside it. *Gen* only generates the behaviours defined in *SLTS*, i.e., with the help of nondeterminism.

## 4 The Basics of Our Formalism

### 4.1 CSP

A thorough treatment of CSP can be found in [13]. Here we give only a brief introduction to the dialects of the CSP language used in this paper. To help with the intuition of acceptors/generators and to make explicit (at least partially) the fact that specifications are constrained by **Norm**, we use two different dialects: $CSP_{di-IMPL}$ for implementations and $CSP_{di-SPEC}$ for specifications.

Syntax of $CSP_{di-IMPL}$

$$\begin{aligned}
LLP ::= {} & l(e_1, \ldots, e_n) \\
& \mid \sqcap x : X \bullet LLP \mid \sqcap \{LLP_1, \ldots, LLP_n\} \\
& \mid c!x_1 \ldots !x_n \to LLP \mid b \,\&\, LLP \\
& \mid \mathsf{let}\;\; l(z_1, \ldots, z_n) = LLP' \;\mathsf{within}\;\; LLP
\end{aligned}$$

$$P \quad ::= LLP \mid P \,[\![\, chans \,]\!]\, P' \mid P \setminus chans$$

Syntax of $CSP_{di-SPEC}$

$$\begin{aligned}
LLQ ::= {} & l(e_1, \ldots, e_n) \\
& \mid \Box \{LLQ_1, \ldots, LLQ_n\} \\
& \mid c?x_1 \ldots ?x_n : b \to LLQ \mid b \,\&\, LLQ \\
& \mid \mathsf{let}\;\; l(z_1, \ldots, z_n) = LLQ' \;\mathsf{within}\;\; LLQ
\end{aligned}$$

$$Q \quad ::= LLQ \mid Q \,[\![\, chans1 \mid chans2 \,]\!]\, Q'$$

where *chans* is a set of channel names, e.g., $\{c_1, \ldots, c_n\}$, and

$$\begin{aligned}
b \; & ::= true \mid false \mid x = x' \mid arr[x] \mid \neg\, b \mid b \wedge b' \mid b \vee b' \\
e \; & ::= x \mid are \\
are \; & ::= arr \mid \{\} \mid \{x\} \mid X \mid are \cup are' \mid are \cap are' \mid are \setminus are'
\end{aligned}$$

In the above, a data independent type $X$ is assumed, and $x$, $y$, $x_i$ and $y_i$ range over the set (i.e., $\chi$) of variables of type $X$. $arr$ and $arr_i$ range over the set (i.e., $\mathtt{ARR}$) of variables of boolean DI array type. $z$ and $z_i$ range over the set of variables of both types, i.e., range over $\mathtt{Z} = \chi \cup \mathtt{ARR}$. A value expression ($e$) can either be a DI variable ($x$) or a DI array value expression ($are$). Value expressions can only be used in a function call ($l(e_1, \ldots, e_n)$) as actual parameters. A function ($l$) is defined using the let construct. In general, it is assumed that each variable's binding occurrence must be unique in a process expression.

$CSP_{di-IMPL}$ is designed to capture the intuition of pro-active and "talkative" generators; all the choice is made internally and processes only output. As there is no **Norm** requirement on implementations, $CSP_{di-IMPL}$ can enjoy the full power of interface parallel and hiding operators. $CSP_{di-SPEC}$ is designed to capture the intuition of passive and receptive acceptors, so *multipath choice* ($\sqcap \{LLP_1, \ldots, LLP_n\}$ or $\Box \{LLQ_1, \ldots, LLQ_n\}$) is made externally and it only inputs. Its higher level operators are confined by **Norm** to alphabetised parallel. (This can be expressed in terms of the interface parallel operator used in $CSP_{di-IMPL}$, but unlike interface parallel cannot introduce graph nondeterminism.) Other features of the languages are:

– The languages implement the intuition of “communicating sequential process”, where a system (i.e., $P$ or $Q$) consists of a network of sequential processes (i.e., $LLP$s or $LLQ$s) running in parallel. Only low-level operators [15] can be used in $LLP$ and $LLQ$ definition. High-level operators like parallel and hiding are used only in composing the network. It will ensure the finite-control (i.e., finite control states) of any process in the language.
– Replicated internal choice in $CSP_{di-IMPL}$ is used to capture the intuition of *internal* or *unrecorded* variables, which cause (sometimes insuperable) problems to the process of translation to an acceptor if they appear on the left hand side of a refinement check. In the current paper, however, this possibility is banned by (i) of **Norm**.
– Selective input in $CSP_{di-SPEC}$ helps introduce *external variables*, or *recorded variables*, explicitly; that is, variables with values assigned by the environment and therefore “recorded” in its behaviours.
– *Boolean guard* ($b \,\&\, LLP$ or $b \,\&\, LLQ$) and multipath choice are used in place of conditionals for the sake of expressiveness and simplicity.
– The condition $b$ in boolean guards is the conjunction, disjunction and negation of two most basic forms of testing: equality testing ($x = x'$) and array testing ($arr[x]$).

Syntactically, the two languages look very restricted and very different from each other. But (trace-)semantically they are, actually, quite expressive and very close to each other.

– Output can be simulated by selective input in $CSP_{di-SPEC}$. Input can be simulated by output and replicated internal choice (given that we use the trace model) in $CSP_{di-IMPL}$.
– Any fixed-finite data types and their operations can be reduced by branching and instantiation into control structure (c.f. [21] for an actual procedure, where case analysis and mutual recursion are used to reduce value-passing CCS to pure CCS). Although this is not recommended for doing real model checking, it is absolutely legitimate and simplifying when developing the theory of data independence and what is formally decidable.
– A finite collection of DI arrays with contents of fixed-finite types and *map* operations (i.e. mapping a $n$-nary fixed-finite operation onto a set of $n$-tuples which collect their $n$ fixed-finite elements from the same location of $n$ DI arrays [11]) can be reduced to a number of sets (i.e. boolean arrays) and combinations of the 3 basic set operations ($\cup$, $\cap$ and complementation).

## 4.2 Unity

As explained earlier, a Unity program consists of a finite set of variables, their initialisation, and a finite set of guarded commands.

Unity variables are typed. For programs in this paper, besides the DI type and the boolean DI array type of the CSP languages, some fixed-finite types are also allowed to encode the control structures and synchronous communications in CSP, since Unity language itself supports neither.

For control structures, a control variable, $CS$, encodes the control states of a CSP process, which will correspond one-to-one to each node in SLTSs/$SLTS_N$s. For communication, a channel variable, $CN$, is used to record the channel name on which the communication occurs, and a binary flag variable $f$ is used to implement the synchronisation between processes. The report of error by an acceptor after encountering an illegal traces is made through a binary variable, $r$.

The set of assignments in each guarded command are simultaneous. That is, the RHS expressions of the assignments are evaluated first; then the evaluated values are assigned to the LHS variables all at once. If we temporarily ignore the fixed finite types and their operations[9], the formal definition of guarded commands can be given below,

$$cmd ::= \; b \longrightarrow as^{10}$$

where $b$ is a boolean guard as in CSP and $as \;::=\; arr := are \mid x := x' \mid x :=?$. The only construct we need to pay attention to is nondeterministic selection, $x :=?$. It means to pick nondeterministically a value from $X$ and assign it to $x$. It is a form of data nondeterminism. With it we can generate DI values implicitly.

After instantiating $X$ with a concrete type $T$, a Unity DI program becomes an ordinary Unity program, whose semantics is modelled by a concrete state-based system. Each concrete state of the system is identified by a value assignment on the set of program variables. Initialisation is the value assignments identifying the initial (concrete) states $\{s_0^1, s_0^2, ..., s_0^k, ...\}$. The dynamics of a Unity program can be understood through the notion of *runs*.

A run is a finite sequence of (concrete) state transitions starting with a initial state $s_0$,

$$s_0 \xrightarrow{cmd_1} s_1 \xrightarrow{cmd_2} s_2 \; ... \; \xrightarrow{cmd_n} s_n$$

That is, for any state $s_{i-1}$ $(1 \leq i \leq n)$, if the guard of $cmd_i$ is true, and $cmd_i$ is fired, it will transit to state $s_i$. ($n$ can be 0, in which case a run degenerates to a single initial state.)

A Unity program is a closed system; it is graph-deterministic iff it has only one run. That is quite simple, but not very useful. More commonly, we will study the deterministic subsystems of a Unity program.

**Definition 1.** *A subprogram $A$ of a Unity program $S$ is deterministic iff at any point of any run of $S$, the subset of commands belonging to $A$ has at most one member enabled, and the member must not use nondeterministic selection in its assignments.*

So caution should be taken on what determinism it means when we mention a deterministic Unity program.

[9] Fixed-finite types and their predicates/operations are theoretically trivial, although their treatment can clutter our presentation non-trivially.
[10] Note in this context we use vectors interchangeably with sets.

One important property of Unity DI programs is the monotonicity of unreachability with respect to the size of $X$. It is quite obvious: by increasing the size of $X$, nondeterministic selection can pick more values, so it can simulate all the runs of the program with a smaller $X$ instantiation, in particular ones reaching designated control states.

More interestingly, that also implies the monotonicity of determinism in Unity. So we need only to check the determinism of a Unity (sub)program when $X$ is infinite to make sure it will be uniformly deterministic over all possible instantiations of $X$. Generally, when we mention a Unity DI (sub)program is deterministic, we mean it uniformly.

### 4.3 The Symbolic LTS

A SLTS is a data-bearing LTS[11]. For DI systems, it adds to a LTS the following,

- Each node is associated with a set of data variables of DI type or boolean DI array type. $vars(n)$ denotes the set of variables for node $n$.
- Each transition is labelled by a triple of *guard*, *symbolic event*, and *assignments*: $ts ::= (gu, se, as)$. Below is a transition from $m$ to $n$.

$$m \xrightarrow{gu,se,as} n$$

- Free and bound variables in the label of a transition must satisfy the following constraint to make the overall SLTS well-formed:

$$fv(se) \cup fv(as) \subseteq vars(m) \ \wedge \ fv(gu) \subseteq vars(m) \cup bv(se)$$
$$\wedge \ vars(n) \subseteq vars(m) \cup bv(se) \cup bv(as)$$

The possible symbolic events ($se$), guards ($gu$), and assignments ($as$) are:

$$\begin{array}{lll} se & ::= & c\ tt \mid \tau \quad (\ tt ::= \epsilon \mid ?x\ tt \mid !x\ tt\ ) \\ gu & ::= & true \mid false \mid x = x' \mid arr[x] \mid \neg\ gu \mid gu \wedge gu \mid gu \vee gu \\ as & ::= & arr := are \mid x := x' \mid x :=? \end{array}$$

The $x$ in $?x$ of $se$, and the $x$ (or $arr$) in $x := x'$ (or $arr := are$) of $as$ are binding occurrences. All other occurrences of DI or array variables in a label are free occurrences.

The intuitive meaning of a transition $m \xrightarrow{gu,se,as} n$ is,

*If the values of variables in node m and the values of input variables (bound variables) in se satisfy gu, the transition is able to fire, after which the evaluated RHSs of as are simultaneously assigned to the (old or new) LHS variables.*

[11] For each process, it is assumed that its LTS or SLTS has a unique *root* node.

Note that as input variables in $se$ can be constrained in $gu$, SLTS transitions can easily encode the selective input of CSP. One complication it might bring us, however, is: there might be name conflicts in $gu$, since it is possible that $vars(m)$ and $bv(se)$ might intersect. We need to adopt a convention that whenever a name conflict arises, e.g., on variable $x$, the primed variable $x'$ will be used to refer to the $x$ in $bv(se)$. More details on this will be in Section 5.2.1.

An SLTS after instantiating $X$ with a concrete type $T$ becomes a concrete SLTS. Concrete states of such SLTSs consist of two parts: a node name, e.g., $m$, identifying the control state, and a value assignment on $vars(m)$, identifying the data state. A sequence of concrete states connected by concrete transition labels form a run of these SLTSs, where a concrete transition label is a transition label ($ts$) with its symbolic event ($se$) replaced by a concrete event.

A concrete SLTS is a concrete communicating system; its determinism is based on traces, which is a sequence of concrete events implied by a run. Each run implies a trace, which is just the sequence of concrete events in the (concrete) transition labels of the run. Whenever a run implies a trace, we say the run conforms to the trace.

**Definition 2.** *A concrete SLTS is deterministic iff, for any trace $tr$ of the SLTS, the run conforming to it must be unique and does not use any nondeterministic selection in the transition labels.*

Based on a similar argument as in DI Unity, the monotonicity of determinism in SLTSs can also be shown to be true. So when we say a SLTS is deterministic, we mean it uniformly.

In general, it is difficult to devise a complete algorithm to decide the determinism of SLTSs (or Unity DI (sub)programs). But an easy-to-check sufficient condition can satisfy most of our needs.

**Lemma 1.** *A SLTS is deterministic, if the transition labels in the graph use no $\tau$ event or nondeterministic selection, and the sibling transitions either share no channel or are disjoint on their guards.*

A SLTS satisfying the condition is also called normalised, or a $SLTS_N$.

## 5 The Translation

In CSP, a process consists of a network of sequential *LLP*s running in parallel. To translate it from CSP to Unity, we adopt a compositional approach.

Firstly, each component *LLP* is translated to a basic automaton in Unity. Then, these basic automata are composed up by a Unity simulation of CSP parallel and hiding operators. This allows us to create the generator and acceptor we needed.

### 5.1 *Impl* to *Gen*

Translating the implementation to a generator is relatively straightforward. We simply follow the procedure outlined below.

For each component *LLP* in *Impl*, do the following.

**Step 1:** $LLP \implies \text{SLTS}$

| Symbolic Labelled Transition Rules | |
|---|---|
| $c!x \rightarrow LLP \xrightarrow{true, c!x, \{\}} LLP$ | |
| $l(e) \xrightarrow{true, \tau, z:=e} LLP$ | where $l(z) = LLP$ |
| $\sqcap x : X \bullet LLP \xrightarrow{true, \tau, \{x:=?\}} LLP$ | |
| $\sqcap \{LLP_1, ..., LLP_n\} \xrightarrow{true, \tau, \{\}} LLP_i$ | |
| $b \;\&\; LLP \xrightarrow{b, \tau, \{\}} LLP$ | |

Each node $n$ is identified by a process expression *LLP*. The set of variables associated with $n$ is, $vars(n) = fv(LLP)$.

**Step 2:** $\text{SLTS} \implies Gen$

* *Interface with the environment*

To implement the SLTS in Unity, first thing to do is to model communication by shared variables, so *Gen* uses a set of *interface variables* to communicate with the environment. Specifically, the following interface variables need be defined, $CN : \{c_1, ..., c_n\}$ for the channel name, $DC : \text{seq}\, X$ for the vector of data components, and $f : \{cont, test\}$ for a synchronisation flag.

* *Encode control and data*

The control structure of the SLTS needs to be encoded by a control state variable, $CS : \{root, m, n, ...\}$, ranging over the node names of the SLTS. Data variables associated with nodes in the SLTS should accordingly be implemented as Unity data variables of the same type. All the control and data variables are *private variables* of *Gen*.

```
 --------
|f|CN|DCs|
|---------------------
|                    |
|  Private variables |
|____________________|
```

* *Implement transitions*

$$m \xrightarrow{gu, \tau, as} n \quad \text{by} \qquad CS = m \wedge gu \longrightarrow \{CS := n\} \cup as$$

$m \xrightarrow{gu, c!x, as} n$ by
$CS = m \wedge gu \wedge f = cont \longrightarrow \{CS := n, CN := c, DC := x, f := test\} \cup as$

* *Initial states*

$f := cont \quad CS := root$

The implemented *Gen* will be able to generate all the possible traces of the SLTS nondeterministically, in the sense that the value of each communication appears in *CN* and *DC*, and separate events are identified by changes in $f$. A new event is recorded when $f$'s value is changed by the process from *cont* to *test* (it being the duty of the observer to change it the other way once the event has been observed).

Hence, *LLP*s can be translated into basic sequential *Gen*s. Based on these results, we can continue to translate *Impl*, which is a network of *LLP*s, into a composite *Gen*.

**Case 1:** Parallel operator

$P \,[\![\, chans \,]\!]\, P'$, where $P$ and $P'$ have been implemented as $Gen(P)$ and $Gen(P')$.

```
  --------
 |f|CN|DCs|
 |---------------------------
 |      Gen(P|[chans]|P')    |
 |________      ________     |
 |f|CN|DCs|    |f|CN|DCs|    |
 |----------|  |----------|  |
 |  Gen(P)  |  |  Gen(P') |  |
 |__________|__|__________|_|
```

* *Commands*

The following three guarded commands observe the external variables of the two subprocesses and combine/transmit these to appear in the external variables of the combination.

$f = cont \wedge f_P = test \wedge CN_P \notin chans$
$\quad \longrightarrow \{CN := CN_P, DC := DC_P, f_P := cont, f := test\}$
$f = cont \wedge f_{P'} = test \wedge CN_{P'} \notin chans$
$\quad \longrightarrow \{CN := CN_{P'}, DC := DC_{P'}, f_{P'} := cont, f := test\}$
$f = cont \wedge f_P = test \wedge f_{P'} = test \wedge CN_P = CN_{P'} \wedge CN_P \in chans \wedge DC_P = DC_{P'}$
$\quad \longrightarrow \{CN := CN_P, DC := DC_P, f_P := cont, f_{P'} := cont, f := test\}$

* *Initial states*

$f := cont$

**Case 2:** Hiding operator

$P \setminus chans$, where $P$ has been implemented as $Gen(P)$.

```
 --------
|f|CN|DCs|
|-------------------------|
|     Gen(P\chans)        |
|--------                 |
|f|CN|DCs|                |
|----------|              |
|  Gen(P)  |              |
|__________|______________|
```

* *Commands*

The following either transmit or conceal each action of $Gen(P)$ as appropriate.

$$
\begin{aligned}
&f_P = test \wedge CN_P \in chans \longrightarrow \{f_P := cont\}\\
&f = cont \wedge f_P = test \wedge CN_P \notin chans\\
&\qquad \longrightarrow \{CN := CN_P, DC := DC_P, f_P := cont, f := test\}
\end{aligned}
$$

* *Initial states*

$$f := cont$$

## 5.2 *Spec* to Acceptor

This is similar to the translation from *Impl* to a generator. The various cases are given below.

For each component $LLQ$ in *NSpec*, do the following.

**Step 1:** $LLQ \implies SLTS_N$

Due to the determinism requirement on $SLTS_N$s, the symbolic labelled transition rules for specifications need to have one important difference from those of implementations; that is, no $\tau$ event can be generated by the rules. Therefore, the specification rules will be designed to merge all possible $\tau$ transitions with their subsequent visible ones.

Symbolic Labelled Transition Rules

$$c?x : b \to LLQ \xrightarrow{b[x'/x],c?x',\{\}} LLQ$$

$$\frac{LLQ_i \xrightarrow{gu,se,as} LLQ'_i}{\square\{LLQ_1,\ldots,LLQ_n\} \xrightarrow{gu,se,as} LLQ'_i}$$

$$\frac{LLQ \xrightarrow{gu,se,as} LLQ'}{b \,\&\, LLQ \xrightarrow{gu \wedge b,se,as} LLQ'}$$

$$\frac{LLQ \xrightarrow{gu,se,as} LLQ'}{l(e) \xrightarrow{gu[e/z],se[e/z],as[e/z] \cup z' := e} LLQ'} \quad \text{where } l(z) = LLQ$$

The middle two rules of symbolic labelled transitions are quite obvious. The first and last rules need to assume a set $\chi'$ of primed variables for $\chi$ and a set $\mathtt{ARR}'$ of primed variables for $\mathtt{ARR}$ so that for any $x \in \chi$, there is a corresponding

$x' \in \chi'$, and similarly for any *arr*. In the context of a transition $\xrightarrow{gu,se,as}$, all occurrences of new variables introduced by binders in *se* and *as* are renamed to their primed counterparts to avoid name conflicts. Note that due to the unique binding occurrence condition on a process expression, only the variables going out of scope through a (direct or indirect) recursive function call may conflict with new variables. The proper working of this mechanism also depends on **Norm** of *LLQ*, which guarantees that all recursive function calls are either action-guarded or are calling on (recursive) functions that behave like *STOP* or *DIV*.

Each node $n$ in the resulting $SLTS_N$ is identified by a process expression *LLQ*. The set of variables associated with $n$ is, $vars(n) = fv(LLQ)$. No sibling transitions in the $SLTS_N$ share any communication channel.

**Step 2:** $SLTS_N \implies Acp$

* *Interface with the environment*

As in *Gen*, the interface includes $CN : \{c_1, ..., c_n\}$ for the channel name, $DC : \text{seq}\, X$ for the vector of data components, and $f : \{cont, test\}$ for synchronisation. Moreover, *Acp* needs an additional variable to report error in accepting, $r : \{normal, error\}$.

* *Encode control and data*

Also like *Gen*, we need a control state variable, $CS : \{root, m, n, ..., stop\}$, bookkeeping the current node in execution or simply *stop*, and a set of data variables implementing variables associated with each node in the $SLTS_N$. They are all private variables of *Acp*.

* *Implement transitions*

Transition $m \xrightarrow{gu,c?x',as} n$ is translated to two commands:

$$CS = m \wedge f = test \wedge CN = c \wedge gu[DC/x'] \longrightarrow \{CS := n, x := DC, f := cont\} \cup as$$
$$CS = m \wedge f = test \wedge CN = c \wedge (\neg\, gu[DC/x']) \longrightarrow \{r := error, CS := stop\}$$

The above raises an error flag if illegal data arrives along a channel that state $m$ can perform. (We note that the conditions of **Norm** ensure that there is at most one transition of $m$ labelled by each channel $c$.) The following does the same if a communication along a channel appears with no transitions in $m$.

$$CS = m \wedge f = test \wedge CN \notin Chans(m) \longrightarrow \{r := error, CS := stop\}$$

* *Initial states*

$f := cont \quad r := normal \quad CS := root$

After we translate *LLQ*s to *Acp*s, only one step is needed to translate *Spec*, that is, translating alphabetised parallel operator.

```
 __________
|r|f|CN|DCs|
|-------------------------|
|  Acp(Q[alp1||alp2]Q')   |
|__________    __________ |
|r|f|CN|DCs|  |r|f|CN|DCs| |
|----------|  |----------| |
|  Acp(Q)  |  |  Acp(Q') | |
|__________|__|__________|_|
```

* A new private variable $ST : \{testing, resting, stop\}$ and six commands in Unity

$f = test \wedge ST = resting \wedge f_Q = cont \wedge f_{Q'} = cont \wedge CN \in (alp1 \cap alp2) \longrightarrow$
$\{CN_Q.DC_Q := CN.DC, CN_{Q'}.DC_{Q'} := CN.DC, f_Q := test, f_{Q'} := test, ST := testing\}$ $f = test \wedge ST = resting \wedge f_Q = cont \wedge f_{Q'} = cont \wedge CN \in (alp1 \setminus alp2) \longrightarrow$
$\{CN_Q.DC_Q := CN.DC, f_Q := test, ST := testing\}$

$f = test \wedge ST = resting \wedge f_Q = cont \wedge f_{Q'} = cont \wedge CN \in (alp2 \setminus alp1) \longrightarrow$
$\{CN_{Q'}.DC_{Q'} := CN.DC, f_{Q'} := test, ST := testing\}$
$f = test \wedge ST = resting \wedge f_Q = cont \wedge f_{Q'} = cont \wedge CN \notin (alp2 \cup alp1) \longrightarrow$
$\{r := error, ST := stop\}$

$f = test \wedge ST = testing \wedge f_Q = cont \wedge f_{Q'} = cont \wedge r_Q = normal \wedge r_{Q'} = normal \longrightarrow$
$\{f := cont, ST := resting\}$

$f = test \wedge ST = testing \wedge (r_Q = error \vee r_{Q'} = error) \longrightarrow \{r := error, ST := stop\}$

* Initial states:

$f := cont \quad ST := resting \quad r := normal$

## 5.3 Connecting the Generator to the Acceptor

Finally, by letting *Gen* and *Acp* sharing the common variables in their interfaces, we connect them up and obtain the *Prog* needed for unreachability check. Variable $r$ is left open to report error.

```
 ________________________
|                        |
|          Acp           |
|________________________|
|r|f|CN|DCs|
 -|-------------------------
  |                        |
  |          Gen           |
  |________________________|
```

This completes our translation of the problem, $NSpec \sqsubseteq_X Impl$, to another problem, $Prog \models_X Unreachability(error)$. Based on the translation, Theorem 4 in [11] can be transferred to the setting of CSP with arrays.

**Theorem 1.** *For any specification of $CSP_{di-SPEC}$ satisfying* **Norm***, the problem of its trace refinement checking by any implementations in $CSP_{di-IMPL}$ is decidable.*

## 6 Conclusion and Future Work

We have shown how to translate certain forms of CSP trace refinement check in a syntax that allows DI array operations into Unity unreachability. With the exception of the *renaming* operator, which we have excluded for simplicity, it is possible to convert any CSP process description of the type usually run on FDR (i.e., parallel/hiding/renaming combinations of sequential processes) to a generator, after noting the trace equivalence of internal and external choice. As a substantial majority of specifications used with FDR either satisfy **Norm** or can be trivially modified to do so – for **Norm** essentially corresponds to clarity of a specification- this means we can confidently expect that our decidability results will cover many practically important cases of CSP checks involving arrays.

Nevertheless, understanding which non-normalised specification processes are capable of being transformed to a finite $SLTS_N$ is important because it will determine the extent to which more general problems of trace refinement can be decided by our methods. The main problem in doing this is understanding the role of unrecorded variables and their relationship to monotonicity. We have investigated this in [16] and introduced the concept of *DI-explicitness* as a way of understanding this. More work is required here.

In general, we believe that the methods described in this paper can be used to extend our results to stable-failures equivalence. Likewise, the work could also be extended to cover the cases of DI arrays that are indexed by one DI type (e.g., $X$) and containing contents of another (e.g., $Y$), or even the cases of multi-dimensional arrays.

The decidability results in this paper are theoretical and sometimes rely on the decision procedures arising from well-structured transition systems [22]. Finding cases where simpler ideas such as threshold calculation [4] will work is important, as is a general investigation of how to translate the decision problems from being theoretically soluble to having access to practical tools that solve them.

**Acknowledgements.** This work was funded by the EPSRC standard research grant 'Exploiting data independence', GR/M32900. Bill Roscoe and Xu Wang were partially supported by a grant from US ONR. Ranko Lazic is also affiliated to the Mathematical Institute, Belgrade, and supported in part by a grant from the Intel Corporation. Xu Wang is now supported by EPSRC grant GR/S11091/01 & GR/S11084/01 and by School of Computer Science, University of Birmingham.

## References

1. Lazic, R.S., Roscoe, A.: Data independence with generalised predicate symbols. In: International Conference on Parallel and Distributed Processing Techniques and Applications. Volume I., Las Vegas, Nevada, USA, CSREA (1999) 319–325
2. Jonsson, B., Parrow, J.: Deciding bisimulation equivalences for a class of non-finite-state programs. In: Symposium on Theoretical Aspects of Computer Science. (1989) 421–433
3. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: Proceedings of the 13th ACM Symposium on Principles of Programming Languages. (1986) 184–193
4. Lazić, R.: A Semantic Study of Data Independence with Applications to Model Checking. PhD thesis, Oxford University Computing Laboratory (1999)
5. Lazić, R., Newcomb, T., Roscoe, A.: On model checking data-independent systems with arrays without reset. Technical Report RR-02-02, Oxford University Computing Laboratory (2002) To appear in the Journal of Theory and Practice of Logic Programming.
6. Lazić, R., Nowak, D.: A unifying approach to data independence. In: Proceedings of the 11th International Conference on Concurrency Theory. Volume 1877 of Lecture Notes in Computer Science., Springer-Verlag (2000) 581–595
7. Lazić, R., Nowak, D.: On a semantic definition of data independence. In: Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications. Volume 2701 of Lecture Notes in Computer Science., Springer-Verlag (2003) 226–240
8. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theoretical Computer Science **256** (2001) 93–112
9. Hennessy, M., Lin, H.: Symbolic bisimulations. Theoretical Computer Science **138** (1995) 353–389
10. Lin, H.: Symbolic transition graph with assignment. In: Proceedings of the 7th International Conference on Concurrency Theory. Volume 1119 of Lecture Notes in Computer Science., Springer-Verlag (1996) 50–65
11. Roscoe, A., Lazić, R.: What can you decide about resetable arrays? (preliminary version). In: Proceedings of the 2nd International Workshop on Verification and Computational Logic, Technical Report, Department of Electronics and Computer Science, University of Southampton, UK (2001)
12. Newcomb, T.: Model Checking Data-Independent Systems With Arrays. PhD thesis, Oxford University Computing Laboratory (2003) To appear.
13. Roscoe, A.: The Theory and Practice of Concurrency. Prentice-Hall (1998)
14. Formal Systems (Europe) Ltd: Failures-Divergence Refinement: FDR2 User Manual. (1999) *http://www.formal.demon.co.uk*.
15. Roscoe, A.W.: Model-checking CSP. In Roscoe, A.W., ed.: A Classical Mind: Essays in Honour of C.A.R. Hoare. Prentice Hall (1994) 353–378
16. Wang, X., Roscoe, A., Lazić, R.: Translating CSP trace refinement to Unity unreachability: a study in data independence. Technical Report RR-03-08, Oxford University Computing Laboratory (2003)
17. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science, Washington, DC (1986) 332–344
18. Valmari, A.: The state explosion problem. In Reisig, W., Rozenberg, G., eds.: Lectures on Petri Nets I: Basic Models. Volume 1491 of Lecture Notes in Computer Science. Springer-Verlag (1998) 429–528

19. Alur, R., Henzinger, T.: Computer-aided verification: An introduction to model building and model checking for concurrent systems. Draft (1999)
20. Milner, R.: Communicating and Mobile Systems: the $\pi$-calculus. Cambridge University Press (1999)
21. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
22. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science **256** (2001) 63–92