Programming Research Group

TRANSLATING CSP TRACE REFINEMENT TO UNITY UNREACHABILITY : A STUDY IN DATA INDEPENDENCE

Xu Wang, A.W. Roscoe, and R.S. Lazić

PRG-RR-03-08

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Translating CSP trace refinement to Unity unreachability : a study in data independence

Xu Wang[†], A.W. Roscoe[†], and R.S. Lazić[‡].

[†] Oxford University Computing Laboratory,
Wolfson Buildings, Parks Road, Oxford OX1 3QD, UK.

[‡] Department of Computer Science, University of Warwick,
Coventry CV4 7AL, UK.

xu.wang@comlab.ox.ac.uk,
bill.roscoe@comlab.ox.ac.uk,
ranko.lazic@dcs.warwick.ac.uk.

Abstract

This paper tries to translate trace refinement between CSP processes to unreachability on Unity-like languages. The key concern, however, is on how data independent (DI) variables and DI arrays could be treated in the translation. It proves to be a challenging task since CSP can utilize DI data in some subtle way which cannot be simulated in Unity. To solve the problem, we find an interesting subclass of CSP specifications called *DI-explicit specifications*. This notion of DI-explicitness bears out to be the necessary condition for translatability, based on which a formal translation procedure is formulated for trace refinement checking problem. Using the semantic constructs of partitions and decorations, the translation is proved to be correct and some extensions are discussed. Overall, this paper is a first step to build a bridge from our recent DI array work in Unity-like languages to our previous DI work in CSP. The main contribution of paper lies in identifying the right condition, framework and mathematical constructs to do the work.

Keywords: CSP, Trace refinement, Model checking, Data independence, Reachability, DI array.

1 Introduction

Algorithmic formal verification of hardware and software systems is a very important research area, where two popular schools of study are in the temporal logical and the process algebraic frameworks.

Whereas model-checking control structures is usually effective and efficient, handling data is more challenging. Often the only way of including data into 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. There is a need to improve understanding of the special properties of data to reduce this problem.

In the real practice of formal system verification (e.g., in our use of CSP/FDR), various non-trivial data is usually indispensable to the description of hardware or software systems. An important class where simplifying results are available are *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 other data of the same kind. Thus a data independent type X is nothing more than an abstract "set". As sets of equal size are completely substitutable via bijection, only the size of X matters.[1]

Typical examples of data independence are, types of data stored in buffer/database or data transmitted by protocols (equality testing is used for data consistency check) or types of address/pointer in a cache/memory (equality testing is used to check if two pointers point to the same location).

[1] It is sometimes possible to weaken this in carefully controlled ways, such as allowing symbols representing functions from the DI type to a known finite type such as the booleans[8].

Theories of data independence have been studied in the past from the process algebra [3] and temporal logic [17] perspectives. Our research has explored both perspectives [4, 5] and we have formulated a unifying semantic framework to capture data independence [6, 7].

While some work on data independence assumes the infinity of DI types[3, 17], no such requirement is present in our work. Actually, our work can handle an infinite families of parameterized systems, where the parameter X can be arbitrary finite size or infinity. The verification problem we face is more than just uniform verification of parameterized systems.

Data independence is also related to another important technique in analyzing data-bearing system: symbolic labelled transition systems [2, 9].

A symbolic LTS tries to represent, in a finite graph, a potentially infinite system caused by 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 : each symbolic state represents an equivalence class of concrete ones.

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 tractability of large state space, abstraction of unbounded structures (like queues, stacks, linear integer, and parameterized 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 result we want to borrow are our recent decidability results on DI arrays[13, 5].

Secondly, in practical applications, both approaches have relative merits and drawbacks. The benefits in process algebraic framework are: unified language, compositionality, and stepwise refinement, while the benefits of temporal logic approach are: abstract property and being liveness/fairness friendly. 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. And the most direct way to show that is by a translation between the checking problems of the two frameworks.

Thus, the following paper will aim to give a two-step translation procedure that could, through an intermediate SLTS, automatically convert a CSP DI refinement checking problem to an equivalent Unity DI model checking problem. And the property of interest here is unreachability.

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

The reason why we only consider reachability/unreachability is because,

- Reachability is a powerful concept. It is well-known that general safety properties can be reduced to the reachability problem.
- By limiting to reachability, decidability results on more powerful DI arrays [13] are achievable, which have great potential for practical applications.
- Reachability gives us a much simpler solution to the translation problem, in line with the familiar experience that safety properties are frequently more amenable to verification than liveness ones.

2 The difficulties and this paper

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

The Unity fragment used in [13, 5] is a much simpler language than CSP. In essence, its programs consist of a finite set of variables, an initialization of the variables, and a finite set of guarded multiple assignments of form, $boolean\_expression \longrightarrow \{assignment\}$[2].

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

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

[2]Note in this paper sometimes vector is used interchangeably with set.

Similar properties are not enjoyed by CSP refinement checking (i.e., for any of the three major models). In CSP we even can construct (*Spec*, *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 $Spec \sqsubseteq Impl$ are inherently not translatable to $Prog \models Unreachability$

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

Some CSP programs can use DI variables to simulate the effects of limited form of arrays.

Therefore, in translating refinement of CSP programs, very often the algorithm will need to invent array variables dynamically on the Unity side. A good example of that is when checking a nondeterministic register, *NDREG*.

$$\begin{array}{lll} NDREG & \equiv & \mathsf{let}\ NR(x) = out!x \rightarrow NR(x) \\ & & \qquad\qquad\qquad \Box \\ & & \qquad\qquad\qquad in?y_0 \rightarrow \text{if } x = y_0 \text{ then } NR(x) \text{ else } NR(x) \sqcap NR(y_0) \\ & & \mathsf{within}\ in?x_0 \rightarrow NR(x_0) \end{array}$$

The parameter variable x above encodes a boolean array (i.e., a set). A more intuitive reformulation using arrays is,

$$\begin{array}{lll} NDREG & = & \mathsf{let}\ NR(arr) = out\$x : arr[x] \rightarrow NR(\{x\}) \\ & & \qquad\qquad\qquad\quad \Box \\ & & \qquad\qquad\qquad\quad in?y_0 \rightarrow \text{if } arr[y_0] \text{ then } NR(arr) \text{ else } NR(arr \cup \{y_0\}) \\ & & \mathsf{within}\ in?x_0 \rightarrow NR(\{x_0\}) \end{array}$$

The syntax used for the example is mainly taken from $CSP_{di-SPEC}$ in Section 4.1. The only additions are the use of binary internal choice and the use of selective nondeterministic selection ($out\$x : arr[x]$), which is a dual of selective input in $CSP_{di-SPEC}$.

This paper will try to overcome or circumvent them in order to give a translation for as large a subset of DI refinement checking problems as possible. Specifically, Section 3 will illustrate the general idea and procedure we use for the translation. Section 4 introduces the basic formalisms, definitions and theorems used in the paper, while Section 5 starts to develop the translation formally.

The translation is divided into two parts. In Section 5.1 *Impl* is translated to a Unity program in the form of a *generator*; it is relatively easy. In Section 5.2, *Spec* is translated; it is somehow more complex. We first discuss the notion of *DI-explicit specifications* and answer the question "within the limit of monotonicity, what specifications can be translated to Unity?" Then, the notion of *acceptors* is introduced and its relationship with determinism explained. Finally, the procedure for translation is presented in detail, and the resulting acceptor Unity program is connected to the generator Unity program in Section 5.3 to complete the translation and show some initial benefits.

In Section 6, we discuss possible improvements on the results in the previous section, and try to extend it to the case where no *norm* condition is assumed on *Spec*.

Section 7 concludes this paper, pointing 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 which of its states are required to be unreachable.

Our initial idea about a possible solution comes from the refinement checking procedure used in FDR. Later, as only the trace model is treated in the first step, we realized that it can be simplified and presented in an automata-theoretical way following [16].

The idea is to construct *Prog* to implement a partial refinement checking on (*Spec*, *Impl*) by exploiting nondeterminism and the fact that in this paper we are only considering the trace model.

- The *Impl* is implemented in Unity as a nondeterministic behaviour generator, *Gen*, which generates 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 behaviour acceptor, *Acp*, which accepts the traces of *Spec*.

- Run *Gen* and *Acp* in parallel. *Gen* outputs an event and waits. *Acp* inputs it and sees if it is precisely acceptable. If accepted, *Acp* signals *Gen* to continue. Otherwise, an *error* occurs. (Note the two automata perform check interactively, rather than on a word-by-word basis, c.f. section 2.5 of [11]).

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

  Thus the overall program *Prog* is, $Prog = Gen \parallel Acp$.

- Model checking to verify unreachability of *error* on *Prog*

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

Since CSP safety properties essentially coincide with trace refinements, this idea transforms them to Unity unreachability.

3.2 The procedure

Unity is a state-based language. Its states are described by a finite set of variables; even control states are encoded into variables. Its semantics can be naturally captured by a SSTS (Symbolic State Transition System).

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

But the difference between SSTSs and SLTSs is not great. They can easily be translated to-and-fro just as do plain STSs and LTSs [15]. In the rest of this paper, therefore, only SLTSs will be explicitly used.

Our translation procedure will have two parts: one for *Impl* and one for *Spec*.

a) Translating *Impl* to a generator is relatively easy

$$Impl \quad \Longrightarrow \quad SLTS \quad \Longrightarrow \quad Gen$$

   In the first step, *Impl* is translated to a nondeterministic SLTS, which can then be implemented in the second step as a Unity nondeterministic *Gen*.

b) Translating *Spec* to an acceptor is more difficult. To concentrate on the problem of data independence, let us assume the norm condition on *Spec* and start with *NSpec*. The reason behind that will be explained later.

$$(Spec \quad \Longrightarrow_{Norm}) \quad NSpec \quad \Longrightarrow \quad SLTS_N \quad \Longrightarrow \quad Acp$$

   NSpec will be translated to a deterministic SLTS, i.e., a $SLTS_N$. Then, the second step constructs a deterministic Unity *Acp* accepting the behaviour in the $SLTS_N$ while monitoring behaviour outside it.

The norm condition[3] on CSP specifications is,

i) *No hiding or renaming.*
ii) *Only alphabetized parallel.*
iii) *Each branch of a multi-path choice is disjoint with others on the channels it will use for the next communication.*

NSpec is a specification satisfying the norm condition.

[3]The norm condition here is a relaxation of the norm conditions in [4, 12] by allowing replicated choices over DI types. It also extended (as did in [12]) the norm condition in [4] by allowing alphabetized parallel.

4 Basics of formalism

4.1 CSP

The most thorough treatment of CSP can be found in [12]. Here included is only a brief explanation of the variants of CSP language used in this paper. Actually, to help with the intuition of acceptors/generators and to make explicit (at least partially) the fact that specifications are constrained by the norm condition, two different variants, $CSP_{di-IMPL}$ for implementations and $CSP_{di-SPEC}$ for specifications, are used in this paper.

Syntax of $CSP_{di-IMPL}$

$$\begin{array}{lll} LLP & ::= & l(e_1, \dots, e_n) \\ & & \mid \sqcap x : X \bullet LLP \mid \sqcap \{LLP_1, \dots, LLP_n\} \\ & & \mid c!x_1 \dots !x_n \rightarrow LLP \mid b \,\&\, LLP \\ & & \mid \mathsf{let}\ \ l(z_1, \dots, z_n) = LLP'\ \ \mathsf{within}\ \ LLP \\ \\ P & ::= & LLP \mid P \,[\![\, chans \,]\!]\, P' \mid P \setminus chans \end{array}$$

Syntax of $CSP_{di-SPEC}$

$$\begin{array}{lll} LLQ & ::= & l(e_1, \dots, e_n) \\ & & \mid \sqcap x : X \bullet LLQ \mid \Box \{LLQ_1, \dots, LLQ_n\} \\ & & \mid c?x_1 \dots ?x_n : b \rightarrow LLQ \mid b \,\&\, LLQ \\ & & \mid \mathsf{let}\ \ l(z_1, \dots, z_n) = LLQ'\ \ \mathsf{within}\ \ LLQ \\ \\ Q & ::= & LLQ \mid Q \,[\![\, chans1 \mid chans2 \,]\!]\, Q' \end{array}$$

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

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

In the above, a data independent type X is assumed, and x, y, x_i and y_i range over the set (i.e., χ) of variables of type X. arr and arr_i range over the set (i.e., $\mathtt{ARR}$) of variables of boolean 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 ($l(e_1, \dots, e_n)$) 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 it only outputs. As there is no norm condition 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* ($\Box \{LLP_1, \dots, LLP_n\}$ or $\Box \{LLQ_1, \dots, LLQ_n\}$) is made externally and it only inputs. Its higher level operators are confined by the norm condition to alphabetized parallel. 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., LLPs or LLQs) running in parallel. Only low-level operators [14] 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 both languages) is used to capture the intuition of internal variables.
- Selective input in $CSP_{di-SPEC}$ helps define external variables explicitly (i.e., variables with values assigned by the environment), distinguishing them syntactically from internal variables.
- *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 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 instantiating into control structures (c.f. [10] 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 theory of data independence.
- DI arrays with elements of fixed-finite type and with *map* operation (i.e. mapping a n-nary fixed-finite operation onto tuples of n elements, where they are collected from n arrays at the same location [13]) can be reduced to sets (i.e. boolean arrays) and 3 basic set operations.

4.2 Unity

As explained in Section 2, a Unity program consists of a finite set of variables, an initialization of these, and a finite set of guarded commands.

Variables are typed. Besides the DI type and the boolean DI array type of the CSP languages, Unity also has some fixed-finite types to encode control structure and implement synchronous communications of CSP programs, as Unity language itself supports neither.

For control structure, a control variable, CS, is used to encode the finite control states of CSP programs, which will correspond one-to-one to the symbolic states in SLTSs/$SLTS_N$s. For communication, a channel variable, CN, is used to record channel name on which the communication occurs, and a binary flag variable f is used to implement the synchronization between processes. Even the final report of error by the acceptor encountering an illegal trace needs to be made through a binary variable, r.

The assignments in each guarded command are simultaneous. That is, the RHS of the assignments in the set are evaluated first, and then the evaluated values are assigned to LHS variables. If we temporarily ignore the fixed finite types and their operations[4], the formal definition of the commands can be given as,

$$b \longrightarrow \vec{as} \quad (\text{where} \;\; as \;::=\; arr := are \mid x := x' \mid x :=?)$$

The only construct we need to pay attention to is nondeterministic selection, $x :=?$. It means to nondeterministically pick a value from X and assign it to x. It is a form of data nondeterminism. Therefore, although explicit values of a data independent type cannot be listed in a program, we can generate their values implicitly using data nondeterminism.

A Unity program is a state-based system. Each concrete state is identified by a value assignment to the set of program variables. Initialization is the value assignment identifying the initial state. The dynamics of a Unity program can be understood through the notion of *run*.

A run is a finite sequence of state transitions starting from initial state s_0,

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

That is, for any state s_{i-1}, the guard of cmd_i is true, and if cmd_i is fired, it will transit to state s_i.

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

Definition 4.1. A subprogram A of a Unity program S is deterministic iff at any point of any run of S, the subset of commands of A can only have one member enabled, and the member must not use nondeterministic selection.

[4] Fixed-finite types and their predicates/operations are theoretically trivial, although they can clutter our presentation non-trivially.

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

One important property of Unity programs for data independence is the monotonicity of unreachability w.r.t. the size of X. It is quite obvious: by increasing the size of X, nondeterministic selection can pick more values, so it can at least simulate all the runs of the program with a smaller X instantiation.

More interestingly, that also implies the monotonicity of determinism in Unity. So we need only to check the determinism of Unity (sub)programs when X is infinite to know it is uniformly true over all X.

4.3 The symbolic LTS

A SLTS is a data-bearing LTS[5]. It adds to a LTS the following,

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

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

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

  $$fv(gu) \cup fv(se) \cup fv(\vec{as}) \subseteq vars(m) \cup bv(se) \quad \wedge \quad vars(n) \subseteq vars(m) \cup bv(se) \cup bv(\vec{as}).$$

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

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

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

> *If the values of variables in node m and the values of inputted variables (bound variables) in se satisfy gu, the transition is fired and $\vec{as}$ are simultaneously assigned to a set of old or new variables.*

Note that as inputted variables in se can be constrained in gu, SLTS transitions can easily encode the selective input of CSP. One complication it might brings 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 co-variable $\overline{x}$ will be used to refer to the x in $vars(m)$. More details on this will be in the latter half of Section 5.2.5.

A concrete state of the system denoted by a SLTS consists 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 concrete transition of the system consists of a concrete event and a set of assignments.

A sequence of concrete states connected by concrete transitions form a run. Each concrete run implies a concrete trace, which is just the sequence of concrete events in the transitions of the run. Whenever some run has a certain trace, we say the run conforms to the trace.

A SLTS is a communicating system; its determinism is actually based on its traces.

Definition 4.2. A SLTS is deterministic iff for any trace tr of the SLTS, the run of the SLTS conforming to tr must be unique and the run does not activate any nondeterministic selection on its way.

SLTSs are no more expressive than Unity, as will be shown by its direct translation to Unity. Based on a similar argument as in Unity, the monotonicity of determinism in SLTSs can also prove true.

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

Theorem 4.3. *A SLTS is deterministic, if the labels in the graph use no τ event or nondeterministic selection, and the sibling transitions either share no channel or be disjoint on their guards.*

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

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

4.4 Partitions and equality testing boolean expression

Equality testing is the only means a DI program can exploit to derive information from DI data. Without it, data independent systems would be much less expressive and its analysis greatly simplified — the parameterized verification problem can usually be reduced to the problem of verifying the system with X instantiated to types of size 1 or 2 [4]. But with equality testing, a lot of challenging problems will face us. The analysis of equality testing requires the following basic definitions and theorems.

An equality testing boolean expression is:

$$et ::= true \mid false \mid x = y \mid et \wedge et' \mid et \vee et' \mid \neg\, et$$

where $fv(et)$ is the set of DI variables used in expression et. We interpret such an expression w.r.t. an alphabet, $alp : \mathbb{P}(\chi)$, using *partitions.* That is, an expression et is interpreted to be a set of partitions on alp via a semantic function, $[\![et]\!]_{alp} : \mathbb{P}(PAR)$.

A partition, $p : PAR$, on alphabet alp is a set of nonempty disjoint subsets of alp. So $PAR = \mathbb{P}(\mathbb{P}(\chi))$. Each subset is a *block* of the partition and their union equals alp. Formally, $[\![et]\!]_{alp}$ is called the *semantics* of et w.r.t. alp.

Each partition on alp implies a truth-value assignment to the set of equality tests, $\{`x = y' \mid x, y \in alp\}$.

- *$x = y$ is true iff x and y are in the same block.*
- *$x = y$ is false iff x and y are in different blocks.*

A partition on alp is a *model* of the expression et iff the truth assignment it implies maps et to $true$. So alp needs be a superset of $fv(et)$ to make a model. An expression et is *satisfiable* iff there exists a model of et. An expression et is *valid* iff all possible partitions on some alp ($\supseteq fv(et)$) are models of et.

A particular partition p on alp is in $[\![et]\!]_{alp}$ iff after applying on et the truth assignment it implies, et is reduced to a satisfiable expression. Especially, when every partition in $[\![et]\!]_{alp}$ can reduce et to a valid expression, the semantics is called *complete* and alp is a *complete alphabet* of et.

Lemma 4.4. *Let alp is a superset of $fv(et)$, p is a model of et iff $p \in [\![et]\!]_{alp}$.*

For an expression et, $fv(et)$ is also called its principal syntax alphabet (pna), and $[\![et]\!]_{pna(et)}$ usually is abbreviated to $[\![et]\!]$, which is the principal semantics of et. Noteworthy, however, is the principal semantics of $true$ and $false$. It is quite obvious that $[\![false]\!]$ should be an empty set of partitions. But $[\![true]\!]$, interestingly, is a singleton of one empty partition.

Completeness has the good property that if any alp and alp' are complete alphabets of et, their intersection is also complete. Therefore, there exists a least complete alphabet for any expression et. The principal semantics alphabet (pma) of an expression et is defined to be the least complete alphabet of et.

Theorem 4.5. *$et \Leftrightarrow et'$ iff $[\![et]\!]_{pma} = [\![et']\!]_{pma}$*

The most important operation on partitions is *hiding.* For a partition p on alp, hiding a $\Delta : \mathbb{P}(\chi)$ from p give us another partition q on $alp \setminus \Delta$, which is just like p but with all the variables in Δ removed from every block of p (if any block becomes empty, the block is removed too). It is written as, $q = p \setminus \Delta$.

Hiding can be promoted to operate on a semantics, $sem : \mathbb{P}(PAR)$, by simply mapping it to every partitions in the semantics, i.e., $sem \setminus \Delta$.

Lemma 4.6. $[\![et]\!]_{alp} \setminus \Delta = [\![et]\!]_{alp\setminus\Delta}$

Other important operations are the *conjunction* of two semantics, $\& : (\mathbb{P}(PAR) \times \mathbb{P}(PAR)) \to \mathbb{P}(PAR)$,

$$\begin{aligned} &sem_{alp1} \ \&\ sem_{alp2} = \{p_{alp1\cup alp2} \mid \exists\, p_{alp1} \in sem_{alp1}, p_{alp2} \in sem_{alp2} \bullet \\ &\quad p_{alp1\cup alp2} \setminus (alp2 \setminus alp1) = p_{alp1} \wedge p_{alp1\cup alp2} \setminus (alp1 \setminus alp2) = p_{alp2}\} \end{aligned}$$

and the *characteristic expression function*, CF, which takes a semantics sem as input and yields its characteristic equality-testing boolean expression et as output.

Lemma 4.7. $[\![et \wedge et']\!] = [\![et]\!] \ \&\ [\![et']\!]$

Theorem 4.8. $CF([\![et]\!]) \Leftrightarrow et$

The most important property of a semantics is its *non-expanding* property relative to Δ.

Definition 4.9. A semantics sem is non-expanding relative to Δ iff for any $p \in sem$, there exists a p' in sem such that $\#p' = \#(p' \setminus \Delta)$ and $p \setminus \Delta = p' \setminus \Delta$.

4.5 Decorated partitions and extended boolean expression

Equality testing boolean expressions are only a restricted form of the boolean expressions to be used in the rest of this paper. More generally, array testing and *array-assignment style equations* are the other constructs required if a fully general *extended boolean expression* is needed.

$$eb ::= b \mid arr = are \mid eb \wedge eb' \mid eb \vee eb'$$

where $fv(eb) = fv_{di}(eb) \cup fv_{arr}(eb)$ is the set of DI variables and array variables used in expression eb.

Array testing has been used in the boolean expressions of boolean guard and selective input. Array-assignment style equations will be used to capture the passing of array values through function calls[6].

Therefore, the extended boolean expressions useful to our work all satisfy a constraint.

Property 4.10. All extended boolean expressions can be transformed to a standard form (seb):

$seb ::= \bigvee\{b_i \wedge asb_i\}$ (where any b_i and b_j are disjoint.)
$asb ::= \bigvee\{ascb_i\}$ (where any $ascb_i$ and $ascb_j$ have the same set of array variables on the LHS of array-assignment style equations.)
$ascb ::= \bigwedge\{arr_i = are_i\}$ (where any arr_i and arr_j on LHS are different and the relation between LHS array variables and their RHS counterparts yields a partial order.)

To analyze extended boolean expressions, we need to extend partitions introduced before in such a way that each block of a partition is additionally decorated with a set of array variables. When a variable arr is in the decoration of a block, it means the DI value denoted by the block is a member of arr.

Formally, each extended boolean expression is interpreted w.r.t. an alphabet, $alp : \mathbb{P}(\chi \cup \mathtt{ARR}) = (alp_{di} : \mathbb{P}(\chi)) \cup (alp_{arr} : \mathbb{P}(\mathtt{ARR}))$. That is, an expression eb is interpreted to be a set of decorated partitions on alp via a semantic function, $[\![eb]\!]_{alp} : \mathbb{P}(DPAR)$.

A decorated partition, $dp : DPAR$ ($DPAR = \mathbb{P}(\mathbb{P}(\chi) \times \mathbb{P}(\mathtt{ARR}))$) on alphabet alp is a set of disjoint decorated subsets of alp_{di}. Each decoration is a subset of alp_{arr}. Each decorated disjoint subset is a *decorated block* of the partition and their union (without decorations) equals alp_{di}. Blocks are allowed to be empty; actually due to different decorations, multiple empty blocks can exist in the same partition[7].

Each decorated partition dp on alp implies a truth-value assignment to the set of equality testing, $\{x = y \mid x, y \in alp_{di}\}$, and the set of array testing, $\{arr[x] \mid x \in alp_{di} \wedge arr \in alp_{arr}\}$.

Each decorated block $dblk = (blk : \mathbb{P}(\chi), dcr : \mathbb{P}(\mathtt{ARR}))$ in dp implies a truth assignment to the set of array-assignment style equations constructed with only variables in alp. Assuming $fv(arr = are) \subseteq alp$, the implication function BL can be defined as,

$$\begin{aligned}
&BL(arr = are) = BL(arr) \Leftrightarrow BL(are)\\
&BL(arr) = true \ \ if \ \ arr \in dcr\\
&BL(arr) = false \ \ if \ \ arr \notin dcr\\
&BL(\{\}) = false\\
&BL(X) = true\\
&BL(\{x\}) = true \ \ if \ \ x \in blk\\
&BL(\{x\}) = false \ \ if \ \ x \notin blk\\
&BL(are \cup are') = BL(are) \vee BL(are')\\
&BL(are \cap are') = BL(are) \wedge BL(are')\\
&BL(are \setminus are') = BL(are) \wedge \neg BL(are')
\end{aligned}$$

A decorated partition dp on alp is a *model* of eb iff, given any $dblk$ in dp, after applying on eb the truth assignment implied by $dblk$ and dp, all the equality testing, array testing, and array-assignment style equations can be reduced and eb becomes *true*. So alp needs be a superset of $fv(eb)$ to make a model. An expression eb is *satisfiable* iff there exists a model of eb. An expression eb is *valid* iff all possible partitions on some alp (superset of $fv(eb)$) are models of eb.

A particular partition dp on alp is in $[\![eb]\!]_{alp}$ iff after applying on eb the truth assignment it implies, eb is reduced to a satisfiable expression. Especially, when all the partitions in a semantics, $[\![eb]\!]_{alp}$, implies a valid expression on eb, the semantics is called *complete* and alp is a *complete alphabet* of eb.

pna, pma and $[\![eb]\!]$ can be defined similarly as above. The only difference is in $[\![true_{eb}]\!]$ which shall be a singleton of one partition with one empty-set decorated empty block.

[6] Obviously, the passing of DI values can be captured by equality testing.

[7] Note that empty blocks do not interfere with disjointness

Operations on partitions and semantics, like *hiding* and *conjunction* can be similar defined as well. We only need to extend $\Delta : \mathbb{P}(\chi)$ to $\Delta : \mathbb{P}(\chi \cup \mathtt{ARR}) = (\Delta_{di} : \mathbb{P}(\chi)) \cup (\Delta_{arr} : \mathbb{P}(\mathtt{ARR}))$ and add the treatment on decorations accordingly.

The *characteristic expression function* needs more attention here. One fact we must notice is that our semantic domain is much richer than extended boolean expressions. Some semantic values may not have any correspondence in extended boolean expressions. One example is $\{\{(\{\},\{arr\}),(\{\},\{\})\}\}$.

The problem occurs largely because of multiple empty blocks and hiding operation. The semantic values with correspondence of boolean expressions will exhibit a decoration pattern on their empty blocks. Initially, all semantic values are derived from a boolean expression; so they possess that pattern. But after applying hiding operation, some non-empty blocks may become empty and be aggregated with other empty blocks. It changes the decoration pattern of empty blocks in the semantic value, and renders it untranslatable.

A *partition skeleton* is a decorated partition with all empty blocks removed; each partition skeleton can be accurately captured by a boolean expression b. For a partition skeleton dps of a semantics sem, there is a *decoration pattern* on its related empty blocks. The pattern consists of a set of *shapes*; each shape comes from a dps-skeletoned decorated partitions in sem, which is the set of decorations on the empty blocks of that partition. Hiding ($\backslash$) and conjunction ($\&$) operations can similarly be defined on patterns.

A possible solution to the above problem can be that we only capture the information in skeletons and discard information in patterns. Thus we will have a function $CF(\{dps\})$ that transforms a semantic value to a b boolean expression. But it is just a partial solution, and is useful only in very limited context. A more general solution can be found by noticing the fact that in most cases of this paper, the rich semantic domain is not absolutely necessary. We can circumvent the problem by putting a restriction on the domain and consider only the subset of semantic values that are *polarizable*.

Polarization of semantic values is based on the definition of two extreme types of pattern values. If the shapes in a pattern are all singletons, the pattern is a *top pattern*. It is easy to see that a top pattern can be captured by an asb expression that do not use any DI variable. If a pattern equals $\mathbb{P}_1(\mathbb{P}(alp_{arr}))$, it is a *bottom pattern*.

Definition 4.11. A semantic value sem is polarizable on $uars_{arr}$ ($\subseteq alp_{arr}$) iff for any partition skeleton dps in sem, its decoration pattern can be represented as a conjunction of the bottom pattern on $uars_{arr}$ and a top pattern on alp_{arr}.

Sometimes we use $sem^{uars_{arr}}$ to denote a semantic value polarizable on $uars_{arr}$.

Theorem 4.12. *All polarizable semantic values have correspondence in extended boolean expressions.*

Although $CF()$ function is only defined for semantic values having correspondence to extended boolean expressions, we will continue to use $CF()$ for arbitrary semantic values in the following because it makes our treatment more uniform and the output of the function will only be used for translating back to semantics values.

The definition of non-expanding property on a semantics is relative to $\Delta : \mathbb{P}(\chi \cup \mathtt{ARR})$, and some other change is in order.

Definition 4.13. A semantics sem is non-expanding relative to Δ iff for any $dp \in sem$, there exists a dp' in sem such that $\#dp' = \#(dp' \setminus \Delta)$ and $dp \setminus \Delta = dp' \setminus \Delta$.

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.

5.1.1 Component *LLP*

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

Step 1: $LLP \implies$ SLTS

Symbolic Labelled Transition Rules

$$c!\vec{x} \to LLP \xrightarrow{true,c!\vec{x},\{\}} LLP$$

$$l(\vec{e}) \xrightarrow{true,\tau,\vec{z}:=\vec{e}} LLP \qquad \text{where } l(\vec{z}) = LLP$$

$$\sqcap\, x : X \bullet LLP \xrightarrow{true,\tau,\{x:=?\}} LLP$$

$$\sqcap\, \{LLP_1, \ldots, LLP_n\} \xrightarrow{true,\tau,\{\}} LLP_i$$

$$b \;\&\; LLP \xrightarrow{b,\tau,\{\}} LLP$$

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

Step 2: 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, $\vec{DC} : \operatorname{seq} X$ for the vector of data components, and $f : \{cont, test\}$ for a synchronization flag.

* Encode control and data

The control structure of the SLTS needs to be encoded by a control state variable, $CS : \{m, n, ...\}$, 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*.

* Implement transitions

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

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

* Initial states:

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

The implemented *Gen* will be able to generate all the possible traces of the SLTS, nondeterministically.

5.1.2 The network of *LLP*s

Step 1: parallel operator

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

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

* Three commands

$$
\begin{array}{l}
f = cont \wedge f_P = test \wedge CN_P \notin chans \longrightarrow \{CN := CN_P, \vec{DC} := \vec{DC}_P, f_P := cont, f := test\} \\
f = cont \wedge f_Q = test \wedge CN_Q \notin chans \longrightarrow \{CN := CN_Q, \vec{DC} := \vec{DC}_Q, f_Q := cont, f := test\} \\
f = cont \wedge f_P = test \wedge f_Q = test \wedge CN_P = CN_Q \wedge CN_P \in chans \wedge \vec{DC}_P = \vec{DC}_Q \\
\quad \longrightarrow \{CN := CN_P, \vec{DC} := \vec{DC}_P, f_P := cont, f_Q := cont, f := test\}
\end{array}
$$

* Initial states:

$$f := cont$$

Step 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)  |               |
|__________|_______________|
```

* Two commands

$$
\begin{array}{l}
f_P = test \wedge CN_P \in chans \longrightarrow \{f_P := cont\} \\
f = cont \wedge f_P = test \wedge CN_P \notin chans \longrightarrow \{CN := CN_P, \vec{DC} := \vec{DC}_P, f_P := cont, f := test\}
\end{array}
$$

* Initial states:

$$f := cont$$

5.2 *Spec* to acceptor

Translating *Spec* to an acceptor is the most difficult part in this paper. Due to the limit of monotonicity, it is known that only a subset of DI refinement checking problem of CSP is translatable to DI unreachability checking in Unity. And based on the idea of generators and acceptors, it has been shown that any *Impl* is translatable to a Unity *Gen*. Consequently, it is easy to deduce that the limit of monotonicity must lie somewhere in the translation of *Spec*.

5.2.1 DI-explicit specifications

Similar to DI Unity programs, the traces of a DI program in CSP are also monotone increasing w.r.t. the size of X. That has been corroborated by our translation of arbitrary $CSP_{di-IMPL}$ programs (and therefore arbitrary $CSP_{di-SPEC}$ programs) to *Gens*.

The reason why refinement checking is non-monotonic is because in some cases the behaviour of *Spec* may change more steeply relative to $|X|$ than that of *Impl*.

But in a real-world sense, "data independence" can be taken to mean that as few as possible assumptions should be made on the size of X when formulating specifications/implementations such that the change of $|X|$ can affect their behaviour in the smallest possible way.

A specification is the correctness criteria for implementations. It is therefore reasonable to impose conditions which enforce the "least affectedness" on them. It is just such a condition we require here.

As $|X|$ decreases, a specification loses its behaviour, but it should lose in a slowest possible way. That is, it loses a trace iff it will otherwise cause a contradiction, i.e., when the number of DI values used in the trace exceeds $|X|$.

Property 5.1 (DI-explicit). If a trace is a behaviour of *Spec* at certain $|T_1|$ and the number of DI values used in it does not exceed $|T_2|$ $(\leq |T_1|)$, then the trace is a behaviour of *Spec* at $|T_2|$.

This kind of good "data independent" specifications are what we called *DI-explicit specifications.*

Theorem 5.2. *Spec is a DI-explicit specification iff for any Impl, if Spec* $\sqsubseteq$ *Impl at* T_1 *and* $|T_2|\leq|T_1|$*, the refinement holds at* T_2 *as well.*

This is quite obvious. A DI-explicit specification is the slowest decreasing in behaviour, so once *Impl* is less than *Spec* at certain $|T_1|$, it will always decrease faster and be less than *Spec* at all smaller $|X|$. It proves one direction of the implication. The other direction of the implication can be proved by utilizing the fact that if *Spec* is not DI-explicit we can construct an implementation that violates the monotonicity in refining *Spec*.

One obvious class of DI-explicit specifications are those specifications satisfying $RecCho_X$ condition [4]. The condition, in essence, requires that all DI variables used in a specification be recorded.

Recorded variables mean that the value of the variables must be initially picked from the history behaviour (i.e., trace) of the process. Syntactically they are those variables introduced in the selective input operator of $CSP_{di-SPEC}$ (i.e., external variables); while unrecorded variables are those introduced in replicated internal choice (i.e., internal variables). Intuitively, we can think of recorded variables as variables controlled by the environment, while unrecorded variables are controlled by *Spec* process itself (i.e., with value internally chosen). So as *Spec* evolves, it collects a set of recorded variables (*rec*) and their value assignments by the environment ($\vec{as}_{rec} : (rec \to X)$[8]). In order to pass a condition *et* in the program, *Spec* can juggle with the values of unrecorded variables (i.e. those in *unrec*) to make *et* true; that is, the passability of *et* is equivalent to the truth of $satisfiability(et[\vec{as}_{rec}])$[9].

Therefore if there is no unrecorded variable in *Spec*, whether *Spec* can pass a condition *et* will solely depend on the values of recorded variables, i.e., on the history trace. We will have,

Theorem 5.3. *All* $RecCho_X$ *specifications are DI-explicit specifications.*

However, unrecorded variables are very useful in expressing a specification just as nondeterminism is very useful in a concise and intuitive FSM description of some real systems. Possessing unrecorded variables also does not necessarily imply *Spec* will not be DI-explicit.

In the following paper, we will work on giving a solution to handling *Spec* with unrecorded variables. As a first step, let us look at some difficulties caused by unrecorded variables.

5.2.2 Acceptor

The responsibility of an acceptor is to identify errors in the behaviour of implementations. It can be nondeterministic by design; sometimes it may accept an input behaviour and sometimes may, contradictorily, refuse it. What is required of an acceptor in theory is only that

- The acceptor will never refuse a correct behaviour.
- For any erroneous behaviour, the possibility must exists for the acceptor to refuse it.

For such an acceptor, the impossibility (i.e., unreachability) of error detection in testing an implementation implies non-existence of erroneous behaviour in that implementation.

Practically, acceptors, like the ideas of tester processes [15] and monitor automata [1], are just yet another re-formulation of a basic algorithm in deciding language containment between two finite automata. The algorithm has two steps:

- The first step is to determinize *Spec* and calculate its complement, $\overline{Spec}$.
- The second step is to check the emptyness of the intersected language, $\overline{Spec} \cap Impl$

To translate *Spec* to an acceptor corresponds to the first step, while unreachability checking corresponds to the second. As *Spec* need be determinized in translation, our acceptor will be deterministic; that is, it never refuses a correct behaviour while it always refuses any erroneous behaviour.

In this paper, to concentrate on our main problem of data independence, a simplifying assumption has been adopted in section 4.2: *Spec*s are assumed to satisfy the norm condition. The norm condition and normalization are closely related to determinization. To understand it fully, some explanation of nondeterminism in CSP is in order.

[8] For a vector (or set) of assignments, it is usually equivalently represented as a function in this paper.

[9] We use [ ] to mean substitution. When a function f is used as parameter (i.e., $[f]$), it means to substitute x with $f(x)$. When a relation R is used as parameter (i.e., $[R]$), it means to substitute x with any $y \in R(\!|\, x \,|\!)$.

Nondeterminism in CSP usually means extensional nondeterminism[10], which is defined in the extensional semantics of processes. The semantics must be fine-grained enough, like stable failure and failure divergence model, to be able to record that nondeterminism. It is nondeterminism in behaviour.

In checking CSP processes, usually we do not use its extensional semantics directly. On the other hand, various kinds of transition graphs are used, like automata, plain LTSs, symbolic LTSs, or annotated LTSs (i.e., with minimal acceptances [14]). Associated with these transition graphs, there is another kind of nondeterminism, which we called *graph nondeterminism.* It is the kind of nondeterminism we see in a nondeterministic automaton; it is caused by silent transitions or similarly labelled sibling transitions in a graph.

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 annotated LTS.

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 following, whenever nondeterminism, determinism, or determinization is mentioned, it means the graphical one. Moreover, sometimes the special CSP terminology for graph determinism, like the norm condition and normalization[11], are also used.

Therefore, in the case of a SLTS, which is actually a data-bearing LTS, the norm condition on *Spec* implies the determinism of the LTS skeleton of its SLTS. To determinize the SLTS completely (i.e., to transform it to a $SLTS_N$), what is left is just to reduce nondeterminism in the data part.

Nondeterminism in the data part lurks in the unrecorded variables as they incur implicit silent branching. If there is no unrecorded variables, i.e., it is $RecCho_X$, the SLTS will be graphically deterministic (or more accurately a $SLTS_N$). To reduce nondeterminism in data is thus equivalent to eliminate unrecorded variables.

Another way to look at the necessity of eliminating unrecorded variables is to compare the *NSpec* translation with the previous *Impl* translation. Unlike the *Impl* translation, when translating *NSpec* to *Acp* we cannot use nondeterminism (i.e., $x :=?$) to choose values for unrecorded variables; for if a wrong value chosen, the acceptor will be punished: it can deadlock too soon or mistakenly refuse a legal trace.

Eliminating unrecorded variables is not an easy task. Our previous *NDREG* example can show that dynamically invented arrays (at least) may be needed to achieve that. A reasonable conjecture can be that,

Conjecture 5.1. *With necessary advanced data structures (e.g., multi-dimension array), any DI-explicit Spec can be translated and normalized to a* $SLTS_N$.

However, it is clear on the other hand that the theoretical barrier of translating *Spec* to an acceptor is indeed the DI-explicit specification.

Theorem 5.4. *If a specification is translatable to some Unity acceptor, then it is DI-explicit.*

Proof. Using monotonicity of Unity unreachability and Theorem 5.2. □

The other direction of the implication is not a theorem because of Conjecture 5.1.

5.2.3 Unrecorded variable elimination

An interesting fact to notice is that when checking $NSpec \sqsubseteq_{T_{Inf}} Impl$ (i.e., when X is infinite), the unrecorded variables in *NSpec* can be reduced by using the hiding operation defined in Section 4.4. The reason behind that is the following theorem.

Theorem 5.5. *Let* $X = T_{Inf}$, $CF(\llbracket et \rrbracket \setminus unrec) \Leftrightarrow \exists\, \vec{as}_{unrec} : (unrec \rightarrow X) \bullet et[\vec{as}_{unrec}]$

A naive reduction algorithm can be devised that terminates for finite behaviour specifications but may fail for infinite behaviour ones.

It is roughly the same when $|X|$ is finite (i.e. arbitrarily finite size), albeit on a condition that a form of DI-explicitness check is done simultaneously with reduction. That is, in juggling DI values for unrecorded variables to make *et* true, only DI values collected through communication can be used. It ensures the healthiness of the reduction when $|X|$ can be arbitrary finite.

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

[11] Strictly speaking, graph-deterministic graphs only correspond to pre-normal form graphs as it may contains semantically equivalent nodes and is not a complete normal form yet [12].

The check is done dynamically while *NSpec* is reduced and translated to a $SLTS_N$. At each checkpoint, only if the check is passed can the reduction safely proceeds. Once the algorithm terminates, it gives us an accurate answer on *NSpec*'s DI-explicitness, and if answer positive, a $SLTS_N$ as well. However, the algorithm in its simple form terminates only for finite behaviour *NSpec*.

For infinite behaviour processes or even infinite state processes, if a terminating algorithm is required, exact analysis of DI-explicitness and reduction cannot be afforded.

Instead, an approximate analysis/reduction algorithm need be introduced. The algorithm will analyze on an upper-approximation (i.e. a superset) of *NSpec*'s reachable states such that it can give us an accurate positive answer (and its $SLTS_N$), but may give false negatives. A negative answer just means the upper-approximation is not DI-explicit; it may or may not implies the real DI-explicitness of *NSpec*.

5.2.4 Parameter variables, deterministic variables and arrays

Previous classification of DI variables into recorded ones and unrecorded ones is largely syntactical and also incomplete. What are left are the formal parameters of a process function, e.g., l . A parameter DI variable, for example, may be instantiated by a recorded variable in one function call, but may also be instantiated by an unrecorded variable in another call. So parameter variables are somehow dynamic in their recordedness.

To treat parameter variables correctly, we need a more general and semantical notion of "recordedness": deterministic variables.

Definition 5.6. A variable z of a DI CSP program is deterministic relative to a concrete trace tr and an instantiatation T (of X), iff either tr is not possible at T or all concrete runs of the program (at T) conforming to tr will end with z in a unique value. For CSP specifications, this kind of relative determinism is monotonic w.r.t. the size of X. So if a variable is deterministic relative to a trace at T_{inf}, it is uniformly deterministic at all X.

The notion of deterministic variables is so general that even "recordedness" in unrecorded variables can be captured. For instance, although initially an unrecorded variable may have a large value range to choose from, at a later stage of the program it may be constrained to be equal to just one recorded variable, whereupon semantically it is also "recorded".

Deterministic variables are very important, because they can be used to construct the $SLTS_N$. Since recorded variables can be introduced through the bound variables in symbolic events, deterministic unrecorded variables and deterministic parameter variables will need to be introduced into SLTSs through the bound variables in assignments, i.e., assigning deterministic values to them.

So we need to change our goal of reduction now. We only need to reduce nondeterministic (unrecorded/parameter) variables in the specification.

The discussions above avoid the discussion of arrays for simplicity. But, as a matter of fact arrays are not difficult to handle in our work, because it is a kind of parameter variables and determinism can be used to classify them too.

Since no replicated choice is allowed over arrays in our language, the only possible sources of nondeterminism for arrays in our systems are multipath choice and unrecorded variables. The norm condition helps us to prevent nondeterminism induced by the former[12]. If we can further guarantee that the values of array variables are independent to nondeterministic variables, they will be deterministic.

But this independence to nondeterministic variables should not be taken literally. It is perfectly legitimate to use a nondeterministic variable in the value expression of an array without breaking its independence. In the contrary, the independence should be understood semantically, to be related to the healthiness of reducing nondeterministic variables from the array value.

All determinism on variables should be decided dynamically as we execute *NSpec*.

5.2.5 The approximate reduction/translation algorithm ($LLQ \implies SLTS_N$)

Our algorithm will reduce and translate *NSpec* directly to a $SLTS_N$, which can later be usefully translated to *Acp*. The algorithm will also be compositional. It needs only to work at the level of component LLQs since alphabetized parallel composition of basic *Acp*s can be implemented in Unity.

Previously, when translating *Impl*, only a partial execution of *Impl* CSP programs is conducted to build the SLTS. It can be obvious by seeing that only control states (i.e., open process terms) are used in firing the transition rules. It is reasonable for *Impl* because,

[12] It is convenient but not necessary for the work in this paper.

- For DI programs, concrete executions using concrete runs are not very useful since each execution covers only a specific instantiations of X.
- As a SLTS records only variable names associated with each node (which are derivable from the open process terms), not its data states, there is no need to execute the data part.

But the case is different when translating *NSpec*: although the $SLTS_N$ still does not record data states, some variables in data states will be reduced and we need data state information to analyze the healthiness of this reduction (e.g., the preservation of trace-equivalence). So for *NSpec*, the data part needs to be executed, and since we cannot do concrete execution on DI data, our complete execution of *NSpec* must be symbolic.

Symbolic execution of CSP programs is based on the notions of *symbolic state* and *symbolic run*. A symbolic state (ss) consists of an open CSP process term (a LLQ) for control state, a set of dependent variables ($vars$) and a constraint boolean expression (pc) for data state. pc characterizes the set of allowable value assignments on $vars$, which must not be empty. Boolean expressions pcs are a subclass of the extended boolean expressions we defined before. Its formal definition is given below.

$$pc ::= b \mid arr = are \mid pc \wedge pc$$

A symbolic run is a sequence of transition label connected symbolic states, $ss_0 \xrightarrow{ts_1} ss_1 \xrightarrow{ts_2} ss_2 \ldots \xrightarrow{ts_n} ss_n$. Data-wise a transition label ts_i can be thought of as a relation from value assignments on $vars(ss_{i-1})$ to value assignments on $vars(ss_i)$. When, for any i in a run, $ts_i(\!|pc(ss_{i-1})|\!) = pc(ss_i)$, we say the run is an exact symbolic run. When $ts_i(\!|pc(ss_{i-1})|\!) \subset pc(ss_i)$, we say the run is an upper-approximate symbolic run.

Due to the symbolic nature of our execution, $pc(ss_i)$, unfortunately, cannot be a "crisp" boolean expression using only variables from $vars(ss_i)$. Rather, since it is derived from $ss_0, \ldots, ss_{i-1}$ step by step via $ts_1, \ldots, ts_i$ and no concrete values can be used to substitute, $pc(ss_i)$ has to refer to history variables (i.e., snapshot of variables in previous states) to keep necessary information. So, as a trace grows longer unboundedly, pc can accumulate variables unboundedly and generate new symbolic states without stopping.

In the following, in a run, $ss_0 \xrightarrow{ts_1} ss_1 \xrightarrow{ts_2} ss_2 \ldots \xrightarrow{ts_n} ss_n$, all variables are assumed to use index to mark which snapshot of the variables they are referring to; for instance, x^0 refers to the snapshot of variable x ($\in vars(ss_0)$) made at ss_0, and x^k ($1 \leq k < n$) refers to the snapshot of x made at ts_k of the run. A new snapshot needs to be made every time a binding occurrence of the variable occurs; that is, when it is bound to a new value, which happens when x is a member of $bv(ts_k)$. The snapshot will be indexed by the location of this binding, i.e., k for ts_k.

After indexing, the good news is that the assignments $\vec{as}$ in any ts_i can be transformed to a boolean expression $be(\vec{as}) = \bigwedge\{be(as)\}$ since now all assignments are done on fresh variables.

$$be(arr := are) = \ arr = are \qquad be(x := x') = \ x = x' \qquad be(x :=?) = \ true$$

And the boolean expression can be combined with guard gu of ts_i to form the extended guard, $egu = gu \wedge be(\vec{as})$, such that a run now is $ss_0 \xrightarrow{egu_1, se_1} ss_1 \xrightarrow{egu_2, se_2} ss_2 \ldots \xrightarrow{egu_n, se_n} ss_n$.

In some simple cases, part or all of history variables in $pc(ss_n)$ may be reducible without sacrificing the accuracy of $vars(ss_n)$ data space. But, in general, we have to adopt some form of approximation to reduce these out-of-scope variables. Only after these variables reduced can the symbolic execution loop back and yield a finite SLTS.

To analyze symbolic runs of a DI system, we need a new symbolic version of traces: DI traces.

Definition 5.7. A DI trace is a symbolic trace, consisting of a sequence of basic symbolic events, $\langle bse_1, bse_2, \ldots, bse_n \rangle$ (where $bse ::= c\ btt$ and $btt ::= \epsilon \mid .x\ btt$) and a partition p on the set of variables used in the trace (i.e., $vars(dtr) = \bigcup\{fv(bse_i)\}$).

More generally, a DI trace should be understood as denoting an equivalent class of the symbolic traces defined above, since the particular choice of symbols (variable names) for data components is irrelevant to its semantics.

Definition 5.8. Two DI traces dtr and dtr' are equivalent iff there exists a $uars : \mathbb{P}(\chi)$ and two functions $f : vars(dtr) \rightarrow uars$ and $f' : vars(dtr') \rightarrow uars$ such that $p[f]$ and $p'[f']$ are sound[13] and $dtr[f] = dtr'[f']$.

[13] It means after substitution partitions remain to be partitions; no variable can occur in two blocks.

DI traces of a (exact or upper-approximate) run, $ss_0 \xrightarrow{egu_1, se_1} ss_1 \xrightarrow{egu_2, se_2} ss_2 \ldots \xrightarrow{egu_n, se_n} ss_n$, are determined by its *transition sequence*, $ss_0 \xrightarrow{egu_1, se_1} \xrightarrow{egu_2, se_2} \ldots \xrightarrow{egu_n, se_n}$.

Definition 5.9. For any transition sequence $ss_0 \xrightarrow{egu_1, se_1} \xrightarrow{egu_2, se_2} \ldots \xrightarrow{egu_n, se_n}$ and any DI trace dtr, where $dtr = (\langle bse(se_1), bse(se_2), ..., bse(se_n)\rangle \setminus \tau, p)$[14], if $pc(ss_0) \wedge \bigwedge\{egu_i \mid i = 1..n\} \wedge CF(\{p\})$ is satisfiable, dtr is a *behaviour* of the transition sequence. Two transition sequences are trace-equivalent iff their sets of behaviours are identical.

The behaviours of a run is the behaviours of its transition sequence. But it should be noted that given a run (or transition sequence), it is generally difficult to determine its behaviours since it needs the global information of the run (or transition sequence) to do that. A more local but nevertheless "approximate" way to determine behaviours is the following.

Definition 5.10. For an exact or upper-approximate symbolic run $ss_0 \xrightarrow{egu_1, se_1} ss_1 \xrightarrow{egu_2, se_2} ss_2 \ldots \xrightarrow{egu_n, se_n} ss_n$ and a DI trace dtr, where $dtr = (\langle bse(se_1), bse(se_2), ..., bse(se_n)\rangle \setminus \tau, p)$, if $pc(ss_n) \wedge CF(\{p\})$ is satisfiable, dtr is a *potential behaviour* of the run. Any behaviour of a run is a potential behaviour of the run.

In its most basic root, behaviour relation is the inverse of the conforming relation we mentioned before. But as for the symbolic case a run may have multiple traces, it needs to be distinguished from the concrete case where a run conforms to only one trace. For a DI-explicit specification, *Spec*, we even can have: for any run rn and any of rn's behaviour dtr, that dtr has an instantiation tr at some X implies that rn will have an instantiation at X conforming to tr.

Definition 5.11. Two runs (upper-approximate or exact) are *path-equivalent* to each other iff they start with the same ss_0 and go through the same sequence of control states and transitions.

Obviously, any two path-equivalent runs are trace-equivalent, as their transition sequences are the same. Especially if an upper-approximate run is path-equivalent to an exact run, it means the run is not a false path of the program, since any exact run has at least one behaviour.

Path-equivalence and partial execution on control states are just the different sides of the same coin; only control states are required if just transition labels and transition sequences are needed for derivation. A partial run can be interpreted as a coarsest symbolic run where pc for every symbolic state is *true*. It can also be interpreted as a skeleton run for the class of runs path equivalent to it. Different runs in a path-equivalence class differ only in their degree of accuracy on data space.

Similarly, the rules used to derive partial runs in previous section can be used as skeleton for the set of rules we will use to execute *NSpec*. The skeleton needs to be augmented with characterizations on data space. So there is exact rule set as well as approximate rule set.

Proposition 5.12. *Different executions of NSpec will be trace-equivalent if the rule sets they used are derived from the same skeleton rule set.*

Given a set of transition rules, a *symbolic execution* of *NSpec* is the set of runs that can be generate by the transition rules on *NSpec*. Similarly we can have exact execution or approximate execution, depending on the set of transition rules used. The DI traces of an execution is the union of all DI traces in every symbolic run of the execution.

Due to the norm condition, any DI trace dtr of *NSpec* will run in just one way. That is, given a set of transition rules, dtr is the behaviour of only one symbolic run of *NSpec*. Moreover, if it is an exact rule set and dtr ($= (\langle bse(ts_1), bse(ts_2), ..., bse(ts_n)\rangle, p)$) is a behaviour of the run $ss_0 \xrightarrow{ts_1} ss_1 \xrightarrow{ts_2} ss_2 \ldots \xrightarrow{ts_n} ss_n$, $pc(ss_n) \wedge CF(\{p\})$ characterizes the set of reachable data states after dtr. For upper-approximate run, $pc(ss_n) \wedge CF(\{p\})$ is a superset of the reachable data states.

Given *NSpec*, our reduction and translation algorithm will try to do an upper-approximate symbolic execution of the program. DI-explicitness check and nondeterministic variable reduction is conducted on-the-fly based on local symbolic state information as *NSpec* is executed. The upper-approximation needs to satisfy some special conditions to guarantee the soundness of the local DI-explicitness check and the reduction.

The final output of transition labels will be the reduced ones with nondeterministic variables removed. The healthiness of reduction implies that the transition sequences based on the reduced ones will be trace-equivalent to their counterparts based on the original ones.

[14] $bse()$ extracts bse from se, but τ is just mapped to itself.

So if the execution is successful (that is, all checks passed and all reduction done), the outputted $SLTS_N$ will be trace-equivalent to the unreduced SLTS.

Due to the determinism requirement on $SLTS_N$s, the skeleton rules on which *NSpec* is based need to have one important difference from those of *Impl* case; that is, no τ event can be generated by the rules. Therefore, these rules shall be specially designed to merge all possible τ transitions with their subsequent visible conterparts. The information on the τ transitions (guard and assignments), accordingly, need be cached for a while in a new *pg* element of the symbolic state before being used later to generate the compound transitions. *pg* is a boolean expression like *pc*. It can be used to cache assignments on τ transition because we can use the same trick as above to transform assignments to extended boolean expressions. Moreover, to facilitate the reduction and the check, the data part of the symbolic states is extended with one even more element, $det : \mathbb{P}(\chi \cup \mathtt{ARR})$, to bookkeep the deterministic variables in *vars*. So now a symbolic state is,

$$((vars, det, pc, pg), LLQ)$$

Data states are sometimes abbreviated as $RDS = (vars, det, pc, pg)$

For a closed term LLQ, its symbolic execution starts with state $((\{\}, \{\}, true, true), LLQ)$, being conducted according to the following symbolic transition rules.

Symbolic Labelled Transition Rules

$$\frac{((vars \cup \{x\}, \dots), LLQ) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ')}{(RDS, \sqcap\, x{:}X \bullet LLQ) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ')}$$

$$\frac{((\dots, pg \wedge b), LLQ) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ')}{(RDS, b \,\&\, LLQ) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ')}$$

$$\frac{(RDS, LLQ_i) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ_i')}{(RDS, \square\{LLQ_1, \dots, LLQ_n\}) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ_i')}$$

$$\frac{(RDS, LLQ) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ')}{(RDS, \mathsf{let}\ \ l(\vec{z}) = LLQ''\ \ \mathsf{within}\ \ LLQ) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ')} \quad \text{bookkeep } (vars, l(\vec{z}) = LLQ'') \text{ in } ENV$$

$$\frac{((\overline{vars} \cup \vec{z}, \overline{det}, \overline{pc}, \overline{pg} \wedge \bigwedge \vec{z} = \vec{\overline{e}}), LLQ) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ')}{(RDS, l(\vec{e})) \xrightarrow{gu,\, se,\, \vec{as}} (RDS', LLQ')} \quad (vars_l, l(\vec{z}) = LLQ) \in ENV \text{ and } \overline{exp} \text{ abbr. } \overline{exp}^{(vars \setminus \overline{\mathsf{Z}} \setminus vars_l)}$$

$$\frac{(vars \cup \vec{x}, det \cup \vec{x}, pc, pg \wedge b) \overset{gu,\, \vec{as}}{\Longrightarrow} RDS'}{(RDS, c?\vec{x}{:}b \rightarrow LLQ) \xrightarrow{gu,\, c?\vec{x},\, \vec{as}} (RDS', LLQ)}$$

Data State Transition Rules

$$(vars, det, pc, pg) \overset{CF(\{dps\}), AS(\llbracket pc'_\mu \rrbracket_{det'}, det' \setminus det)}{\Longrightarrow} (vars \setminus \overline{\mathsf{Z}}, det', CF(\llbracket pc'_\mu \rrbracket \setminus \overline{\mathsf{Z}}), true)$$

Let us postpone the explanation of data state transition rules to concentrate on the six symbolic labelled transition rule firstly.

The first four rules of symbolic labelled transitions are quite obvious. The fifth rule needs to assume a set $\overline{\chi}$ of co-variables for χ and a set $\overline{\mathtt{ARR}}$ of co-variables for $\mathtt{ARR}$ so that for any $x \in \chi$, there is a corresponding $\overline{x} \in \overline{\chi}$ and similarly for any *arr*. The union of $\overline{\chi}$ and $\overline{\mathtt{ARR}}$ is $\overline{\mathsf{Z}}$. The operation $\overline{exp}^{(uars:\mathbb{P}(\mathsf{Z}))}$ changes all the occurrences of *uars* variables in *exp* to their co-variables, i.e., change to co-variables all the variables going out of scope after the function call. These co-variables will be reduced in the firing of the subsequent sixth rule. Co-variables, in essence, implement a local version of indexing we mentioned before. Localness is what we sought after in the reasoning of behaviours below.

The sixth rule is the only rule that do some "real work". Given an input symbolic state matching its antecedent, it generates a transition consisting of a transition label and an output symbolic state. All other rules do not generate any transition; they just "massage" the input symbolic state into the pattern defined by the sixth rule's antecedent.

Basically, the sixth rule comes from a plain upper-approximate symbolic transition rule like the following,

$$((vars, det, pc, pg), c?\vec{x} : b \rightarrow LLQ) \xrightarrow{pg \wedge b,\, c?\vec{x}} ((vars \cup \vec{x}, DV(pc', det \cup \vec{x}), pc', true) \setminus \overline{\mathsf{Z}}, LLQ)$$
$$(\text{where } pc' = pc \wedge pg \wedge b)$$

$DV(pc', det \cup \vec{x})$ calculates deterministic variables in pc' based on $det \cup \vec{x}$. Approximation is done by reducing (i.e., hiding) co-variables $\overline{\mathsf{Z}}$ in the output symbolic data state, $(vars \cup \vec{x}, DV(pc', det \cup \vec{x}), pc', true)$. However, this reduction must be done with care such that the upper-approximate symbolic states created must satisfy a special property to maintain the soundness of future local DI-explicitness check and nondeterministic variable reduction.

Definition 5.13 (the chaining execution). Given an execution of *NSpec*, if for any state ss_{k-1} ($k \geq 1$) in any run, ... $ss_{k-1} \xrightarrow{pg_k, se_k}$ ..., of the execution, the following holds for any DI trace $dtr = (\langle ..., bse(se_k)...\rangle, p)$,

$$\begin{aligned}
&\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k-1\} \wedge CF(\{p\})\ ) \\
&\wedge satisfiability(\ pc(ss_{k-1}) \wedge pg_k \wedge CF(\{p\})\ ) \\
&\Leftrightarrow satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k\} \wedge CF(\{p\})\ )
\end{aligned}$$

, then the execution is said to be chaining.

Lemma 5.14. *Given a chaining execution and any of its run,* $ss_0 \xrightarrow{pg_1, se_1} ss_1 \xrightarrow{pg_2, se_2} ss_2 \ ... \xrightarrow{pg_n, se_n} ss_n$*, for any* $dtr = (\langle bse(se_1), bse(se_2), ..., bse(se_n)\rangle, p)$ *we have,* $\bigwedge\{satisfiability(\ pc(ss_{i-1}) \wedge pg_i \wedge CF(\{p\})\ ) \mid i = 1..n\} \Leftrightarrow satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..n\} \wedge CF(\{p\})\ )$

With this lemma, the global reasoning needed for behaviour analysis is reduced to a sequence of local reasonings.

But how do we guarantee that our upper-approximation could give us chaining executions?

Definition 5.15 (the local chaining execution). Given an execution and any state transition $ss_{k-1} \xrightarrow{pg_k, se_k} ss_k$ (through which *uars* goes out of scope) in any run, ... $ss_{k-1} \xrightarrow{pg_k, se_k} ss_k$ ... in it, if for all potential behaviour of the run $dtr = (\langle ..., bse(se_k), ...\rangle, p)$, we have,

$$\begin{aligned}
&satisfiability(pc(ss_{k-1}) \wedge pg_k \wedge CF(\{p\})) \wedge satisfiability(pc(ss_k) \wedge CF(\{p\}) \wedge xgu) \\
&\Leftrightarrow satisfiability(pc(ss_{k-1}) \wedge pg_k \wedge CF(\{p\}) \wedge xgu)
\end{aligned}$$

valid for any possible extended boolean expression xgu that uses no variable in $uars$, then the execution is locally chaining.

Theorem 5.16. *All local chaining executions are chaining.*

Proof. For any run $ss_0 \xrightarrow{pg_1, se_1} ss_1 \xrightarrow{pg_2, se_2} ss_2 \ ... \ ss_{k-1} \xrightarrow{pg_k, se_k} ss_k \xrightarrow{pg_{k+1}, se_{k+1}}$ ... in a local chaining execution, it is obvious that the chaining condition is automatically satisfied on ss_0 while it coinciding with the chaining condition on ss_1. Then for any ss_k,

$$\begin{aligned}
&\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k\} \wedge CF(\{p\})\ ) \\
&\wedge satisfiability(\ pc(ss_k) \wedge pg_{k+1} \wedge CF(\{p\})\ ) \\
&\Rightarrow \qquad (duetoupper - approximation) \\
&\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k-1\} \wedge CF(\{p\})\ ) \\
&\wedge satisfiability(pc(ss_{k-1}) \wedge pg_k \wedge CF(\{p\})) \\
&\wedge satisfiability(\ pc(ss_k) \wedge pg_{k+1} \wedge CF(\{p\})\ ) \\
&\Rightarrow \qquad (let\ xgu = pg_{k+1}) \\
&\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k-1\} \wedge CF(\{p\})\ ) \\
&\wedge satisfiability(pc(ss_{k-1}) \wedge pg_k \wedge CF(\{p\}) \wedge pg_{k+1}) \\
&\Rightarrow \qquad (let\ xgu = pg_{k+1} \wedge pg_k) \\
&.... \\
&\Rightarrow satisfiability(\ pc(ss_0) \wedge CF(\{p\})\ ) \wedge satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k+1\} \wedge CF(\{p\})\ )
\end{aligned}$$

The implication in the other direction is obvious. □

Firstly, let us define the deterministic-variable calculation function $DV(pc, det) : \mathbb{P}(\overline{\mathsf{Z}} \cup \mathsf{Z})$ and the reduction operation, $(vars, det, pc, true) \setminus \overline{\mathsf{Z}}$.

Definition 5.17. Given a pc and a det, a variable $z : (\overline{\mathsf{Z}} \cup \mathsf{Z})$ is in $DV(pc, det)$ iff $\# [\![pc]\!]_{det \cup \{z\}} = \# [\![pc]\!]_{det}$ and $[\![pc]\!]$ is non-expanding relative to $\{z\}$ if z is a DI variable.

$$(vars, det, pc, true) \setminus \overline{\mathsf{Z}} = (vars \setminus \overline{\mathsf{Z}}, det \setminus \overline{\mathsf{Z}}, CF(\ [\![pc]\!] \setminus \overline{\mathsf{Z}}), true)$$

To make the executions local chaining, the following conditions need to be satisfied by the output symbolic data state and the hiding.

C1 $[\![pc]\!]$ is non-expanding relative to the nondeterministic co-variables (i.e., $(vars \cap \overline{\mathrm{Z}}) \setminus det$).

C2 Nondeterministic proper variables must be independent to deterministic co-variables and no information concerning nondeterministic proper variables can be lost. That is,

$$[\![pc]\!]_{(vars \cap \mathrm{Z}) \cup det} = [\![pc]\!]_{vars \cap \mathrm{Z}} \ \& \ [\![pc]\!]_{det}$$

When *NSpec* is executed and the above rule fired, these conditions need to be checked dynamically. Only if they are passed can the execution safely continue. If a run passed all the check on its way, it is a *successful* run, otherwise it is unsuccessful.

If all runs of *NSpec* are successful, we say the execution of *NSpec* is successful.

Theorem 5.18. *Any successful NSpec execution is local chaining.*

The sixth rule is more than a plain upper-approximate symbolic transition rule. Besides the function of reducing co-variables from output symbolic data state, it also does DI-explicitness check locally and reduces nondeterministic variables from the extended guards *pg* of transition labels, which preserves the behaviour of the original ones.

So an original run $ss_0 \xrightarrow{pg_1, se_1} ss_1 \xrightarrow{pg_2, se_2} ss_2 \ ... \xrightarrow{pg_n, se_n} ss_n$ is reduced to $ss_0 \xrightarrow{egu_1^{det}, se_1} ss_1 \xrightarrow{egu_2^{det}, se_2} ss_2 \ ... \xrightarrow{egu_n^{det}, se_n} ss_n$. The following lemma will help ensure the trace-equivalence between the two runs.

Lemma 5.19. *For any local chaining run,* $ss_0 \xrightarrow{pg_1, se_1} ss_1 \xrightarrow{pg_2, se_2} ss_2 \ ... \xrightarrow{pg_n, se_n} ss_n$*, its reduced version,* $ss_0 \xrightarrow{egu_1^{rec}, se_1} ss_1 \xrightarrow{egu_2^{rec}, se_2} ss_2 \ ... \xrightarrow{egu_n^{rec}, se_n} ss_n$*, and any DI trace,* $dtr = (\langle bse(se_1), bse(se_2), ..., bse(se_n) \rangle, p)$*, if* $satisfiability(pc(ss_{i-1}) \wedge pg_i \wedge CF(\{p\})) \Leftrightarrow satisfiability(pc(ss_{i-1}) \wedge egu_i \wedge CF(\{p\}))$ *for all* $1 \leq i \leq n$*, then the transition sequences,* $ss_0 \xrightarrow{pg_1, se_1} \xrightarrow{pg_2, se_2} \ ... \xrightarrow{pg_n, se_n}$ *and* $ss_0 \xrightarrow{egu_1^{rec}, se_1} \xrightarrow{egu_2^{rec}, se_2} \ ... \xrightarrow{egu_n^{rec}, se_n}$ *are trace-equivalent.*

Proof. For $n = 0$ and $n = 1$, it is trivially true.

Assuming it is also true for $k - 1$, then for $n = k$,

$$\begin{array}{l}
\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{egu_i \mid i = 1..k\} \wedge CF(\{p\})\ ) \\
\Leftrightarrow \text{(assumption for k-1)} \\
\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k-1\} \wedge egu_k \wedge CF(\{p\})\ ) \\
\Leftrightarrow \text{(local chaining with } xgu = egu_k\text{)} \\
\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k-1\} \wedge CF(\{p\})\ ) \wedge satisfiability(pc(ss_{k-1}) \wedge egu_k \wedge CF(\{p\})) \\
\Leftrightarrow \\
\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k-1\} \wedge CF(\{p\})\ ) \wedge satisfiability(pc(ss_{k-1}) \wedge pg_k \wedge CF(\{p\})) \\
\Leftrightarrow \text{(chaining)} \\
\quad satisfiability(\ pc(ss_0) \wedge \bigwedge\{pg_i \mid i = 1..k\} \wedge CF(\{p\})\ )
\end{array}$$

□

The reduction is defined formally below. For any local chaining data state transition[15],

$$(vars, det, pc, pg) \xrightarrow{pg} (...)$$

Its reduced version should be,

$$(vars, det, pc, pg) \xrightarrow{CF(\ [\![pg \wedge pc]\!]_{det})} (...)$$

Let $egu^{det} = CF(\ [\![pg \wedge pc]\!]_{det})$. To have the reduced version satisfy condition $satisfiability(pc \wedge pg \wedge CF(\{p\})) \Leftrightarrow satisfiability(pc \wedge egu^{det} \wedge CF(\{p\}))$ in the lemma, what is needed is the local DI-explicitness check condition.

C3 $pg \wedge pc$ is non-expanding relative to nondeterministic variables (i.e., $vars \setminus det$).

[15] Since the reduction is unrelated to control states and symbolic events, they can be left out and we can have transition rules solely based on data states and extended guards.

The final step is to factor out the assignments implicit in egu^{det} and restore the extended guard to the basic form of guard and assignments. As had been mentioned before, the assignments must be on deterministic variables, or more precisely a superset of the set of newly found deterministic variables (since old ones must have already been assigned).

$DV(pc \wedge pg, det) \setminus det$ certainly is the set of newly found deterministic variables. But the way $DV()$ is used here cannot help us to find all deterministic variables. The reason behind that is that determinism on variables is defined relative to a DI trace. But in the above, $DV()$ is used to analyze directly on $pc \wedge pg$, which usually contains many a DI traces; so $DV()$ could only find the subset of deterministic variables that are uniformly deterministic on all DI traces.

If we want to find all possible deterministic variables in a state transition, we have to branch the rule to do case analysis on individual behaviour.

$$(vars, det, pc, pg) \xrightarrow{CF(\{dp\})} (vars \setminus \overline{\mathsf{Z}}, DV(pc'_{\mu}, det) \setminus \overline{\mathsf{Z}}, CF(\llbracket pc'_{\mu} \rrbracket \setminus \overline{\mathsf{Z}}), true)$$

where $dp \in \llbracket pg \wedge pc \rrbracket_{det}$ and $pc'_{\mu} = pc \wedge pg \wedge CF(\{dp\})$.

Lemma 5.20. *For the successful branching execution of NSpec, $DV()$ can find all the deterministic variables in it.*

A collateral benefit of branching is that it also helps the extraction of assignments, because without branching a similar situation may arise: we could only do assignments on deterministic variables that equal to one value uniformly on all DI traces. Therefore the final rule is below.

Data State Transition Rules

$$(vars, det, pc, pg) \overset{CF(\{dps\}), AS(\llbracket pc'_{\mu} \rrbracket_{det'}, det' \setminus det)}{\Longrightarrow} (vars \setminus \overline{\mathsf{Z}}, det', CF(\llbracket pc'_{\mu} \rrbracket \setminus \overline{\mathsf{Z}}), true)$$

where $dps \in \llbracket pg \wedge pc \rrbracket_{det}$, $pc'_{\mu} = pc \wedge pg \wedge CF(\{dps\})$ and $det' = DV(pc'_{\mu}, det) \setminus \overline{\mathsf{Z}}$.

$AS()$ extracts the assignments on the set of newly found deterministic variables ($det' \setminus det$). If it is a DI variable, it will be assigned by a DI variable in det. If it is an array variable, it will be assigned by an *are* constructed from variables in det.

But $\llbracket pc'_{\mu} \rrbracket_{det'}$ contains a set of similarly-skeletoned partitions, which differ only in their decoration patterns. Newly found deterministic variables need to be uniformly equal to one value across the set of partitions to be able to construct the assignment. Fortunately, this requirement is satisfied by the following lemmas.

Lemma 5.21. *Condition **C3** and $\exists\, sem_{vars}^{det_{arr}} \bullet \llbracket pc \wedge pg \rrbracket_{vars} = (sem_{vars}^{det_{arr}} \,\&\, \llbracket pc \wedge pg \rrbracket_{vars_{di} \cup det_{arr}})$ implies $\exists\, sem_{vars}^{det_{arr}} \bullet \llbracket pc \wedge pg \rrbracket_{vars} = sem_{vars}^{det_{arr}} \,\&\, \llbracket pc \wedge pg \rrbracket_{det}$*

det_{arr} denotes the set of deterministic array variables in $vars$. Condition $\exists\, sem_{vars}^{det_{arr}} \bullet \llbracket pc \wedge pg \rrbracket_{vars} = (sem_{vars}^{det_{arr}} \,\&\, \llbracket pc \wedge pg \rrbracket_{vars_{di} \cup det_{arr}})$ can be proved on all symbolic states in a successful run by induction from the starting state of the run.

Executing *NSpec* according to the above rules and discarding all but det in the symbolic states, finally we will obtain the $SLTS_N$.

5.2.6 An important optimization

Information cannot be lost on nondeterministic variables but can be lost on deterministic variables. So it would always be advantageous if we could convert some deterministic variables to nondeterministic variables.

Sometime, a nondeterministic proper DI variable x may be dependent on some deterministic co-variables, $\overline{x_1}$, $\overline{x_2}$, and $\overline{x_3}$; for example, x can choose to be any of the three but not anything outside them. It breaks condition **C2** and renders the *NSpec* execution unsuccessful. But if we add a deterministic proper array variable arr_x to encode the value range of x, we can keep the information on the value range even after the three co-variables be removed. After associating x with arr_x, x can be independent to the co-variables, which thereafter can be reduced.

So the problem becomes how can the arr_x be added to dissolve the dependence on co-variables. The method is, firstly, adding to the decoration of each block where x is a member, and then trying to decide if arr_x is deterministic. If so, the dependency is dissolvable by arr_x and x_1, x_2, and x_3 are reducible.

arr_x will be associated to x for its lifecycle; it goes out of scope and becomes co-variable iff x does.

Now we can try to use our procedure to translate the *NDREG* example before to a $SLTS_N$.

Firstly, the initial definition is normalized to be,

$$
\begin{aligned}
NNDREG \equiv \ & \mathsf{let}\ NR(x) = out?x_1 : x_1 = x \rightarrow NR(x_1) \\
& \qquad\qquad\qquad \Box \\
& \qquad\qquad in?y_0 \rightarrow \sqcap x' : X \bullet (x' = x \vee x' = y_0)\ \&\ NR(x') \\
& \mathsf{within}\ in?x_0 \rightarrow NR(x_0)
\end{aligned}
$$

Then, we can derive the $SLTS_N$ by executing $NNDREG$ according to the symbolic labelled transition rules we defined before.

To simplify presentation, we can notice that there are seven control states for $NDREG$ process. Let cs_0 denotes the initial state (i.e., the $NDREG$ process itself), and let cs_1 and cs_1', cs_2 and cs_2', and cs_3 and cs_3' denote respectively the states before and after performing $in?x_0$, $in?y_0$ and $out?x_1$, then its execution can be defined below,

$((\{\},\{\}, true, true), cs_0)$ evolves to $((\{\},\{\}, true, true), cs_1)$ (using rule 4)

$((\{\},\{\}, true, true), cs_1) \xrightarrow{true, in?x_0, \{\}} ((\{x_0\},\{x_0\}, true, true), cs_1')$ (using rule 6)

$((\{x_0\},\{x_0\}, true, true), cs_1')$ evolves to $((\{\overline{x_0}, x\},\{\overline{x_0}\}, true, x = \overline{x_0}), cs_2)$ (using rule 5 and 3)

$((\{\overline{x_0}, x\},\{\overline{x_0}\}, true, x = \overline{x_0}), cs_2) \xrightarrow{true, in?y_0, \{x = \overline{x_0}\}} ((\{x, y_0\},\{x, y_0\}, true, true), cs_2')$ (using rule 6)

$((\{x, y_0\},\{x, y_0\}, true, true), cs_2')$ evolves to
$((\{\overline{x}, \overline{y_0}, \overline{x'}, x\},\{\overline{x}, \overline{y_0}\}, true, (\overline{x'} = \overline{x} \vee \overline{x'} = \overline{y_0}) \wedge x = \overline{x'}), cs_2)$ (using rule 1, 2, 5 and 3)

Here we will find that arr_x is needed if we want $\overline{x}$ and $\overline{y_0}$ to be reducible in the next step.

$((\{\overline{x}, \overline{y_0}, \overline{x'}, x, arr_x\},\{\overline{x}, \overline{y_0}\}, true, (\overline{x'} = \overline{x} \vee \overline{x'} = \overline{y_0}) \wedge x = \overline{x'} \wedge arr_x = \{\overline{x}\} \cup \{\overline{y_0}\}), cs_2)$
$\xrightarrow{true, in?y_0, \{arr_x = \{\overline{x}\} \cup \{\overline{y_0}\}\}} ((\{x, arr_x, y_0\},\{arr_x, y_0\}, pc, true), cs_2')$ (using rule 6)
where $pc = CF(\{\{(\{\},\{arr_x\}), (\{x\},\{arr_x\}), (\{\},\{\})\}\})$

$((\{x, arr_x, y_0\},\{arr_x, y_0\}, pc, true), cs_2')$ evolves to
$((\{\overline{x}, \overline{arr_x}, \overline{y_0}, \overline{x'}, x, arr_x\},\{\overline{arr_x}, \overline{y_0}\}, \overline{pc}, pg), cs_2)$ (using rule 1, 2, 5 and 3)
where $pg = (\overline{x'} = \overline{x} \vee \overline{x'} = \overline{y_0}) \wedge x = \overline{x'} \wedge arr_x = \{\overline{arr_x}\} \cup \{\overline{y_0}\}$

$((\{\overline{x}, \overline{arr_x}, \overline{y_0}, \overline{x'}, x, arr_x\},\{\overline{arr_x}, \overline{y_0}\}, \overline{pc}, pg), cs_2)$
$\xrightarrow{true, in?y_0, \{arr_x = \overline{arr_x} \cup \{\overline{y_0}\}\}} ((\{x, arr_x, y_0\},\{arr_x, y_0\}, pc, true), cs_2')$ (using rule 6)

Further execution along this path will give us a loop, so let us pursue the other path in the program.

$((\{x, arr_x, y_0\},\{arr_x, y_0\}, pc, true), cs_2')$ evolves to (using rule 1, 2, 5 and 3)
$((\{\overline{x}, \overline{arr_x}, \overline{y_0}, \overline{x'}, x\},\{\overline{arr_x}, \overline{y_0}\}, \overline{pc}, (\overline{x'} = \overline{x} \vee \overline{x'} = \overline{y_0}) \wedge x = \overline{x'}), cs_3)$

$((\{\overline{x}, \overline{arr_x}, \overline{y_0}, \overline{x'}, x\},\{\overline{arr_x}, \overline{y_0}\}, \overline{pc}, (\overline{x'} = \overline{x} \vee \overline{x'} = \overline{y_0}) \wedge x = \overline{x'}), cs_3)$
$\xrightarrow{\overline{arr_x}[x_1] \vee x_1 = \overline{y_0}, out?x_1, \{\}} ((\{x, x_1\},\{x_1\}, true, true), cs_3')$ (using rule 6)

5.2.7 $SLTS_N \implies Acp$ (LLQ Level)

* Interface with the environment

As in Gen, the interface includes $CN : \{c_1, ..., c_n\}$ for channel name, $\vec{DC} : \mathrm{seq}\, X$ for vector of data components, and $f : \{cont, test\}$ for synchronization. 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, $CS : \{m, n, ...\}$, bookkeeping the node names of the $SLTS_N$, 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 $m \xrightarrow{gu, c?\vec{x}, \vec{as}} n$ is translated to two commands:

$$CS = m \wedge f = test \wedge CN = c \wedge gu[\vec{DC}/\vec{x}] \longrightarrow \{CS := n, \vec{x} := \vec{DC}, f := cont\} \cup \vec{as}$$
$$CS = m \wedge f = test \wedge CN = c \wedge (\neg\, gu[\vec{DC}/\vec{x}]) \longrightarrow \{r := error, CS := none\}$$

For each node m in the $SLTS_N$, there is also an overall refusing command:

$$CS = m \wedge f = test \wedge CN \notin Chans(m) \longrightarrow \{r := error, CS := none\}$$

* Initial states:

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

The implemented Acp can check inputted events against the $SLTS_N$.

5.2.8 The network of LLQs

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

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

$$f = test \wedge ST = resting \wedge f_P = cont \wedge f_Q = cont \wedge CN \in (alp1 \cap alp2)$$
$$\longrightarrow \{CN_P.\vec{DC}_P := CN.\vec{DC}, CN_Q.\vec{DC}_Q := CN.\vec{DC}, f_P := test, f_Q := test, ST := testing\}$$
$$f = test \wedge ST = resting \wedge f_P = cont \wedge f_Q = cont \wedge CN \in (alp1 \setminus alp2)$$
$$\longrightarrow \{CN_P.\vec{DC}_P := CN.\vec{DC}, f_P := test, ST := testing\}$$
$$f = test \wedge ST = resting \wedge f_P = cont \wedge f_Q = cont \wedge CN \in (alp2 \setminus alp1)$$
$$\longrightarrow \{CN_Q.\vec{DC}_Q := CN.\vec{DC}, f_Q := test, ST := testing\}$$
$$f = test \wedge ST = resting \wedge f_P = cont \wedge f_Q = cont \wedge CN \notin (alp2 \cup alp1) \longrightarrow \{r := error, ST := none\}$$
$$f = test \wedge ST = testing \wedge f_P = cont \wedge f_Q = cont \wedge r_P = normal \wedge r_Q = normal$$
$$\longrightarrow \{f := cont, ST := resting\}$$
$$f = test \wedge ST = testing \wedge (r_P = error \vee r_Q = error) \longrightarrow \{r := error, ST := none\}$$

* 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|seq|
  -|--------------------------
   |                          |
   |           Gen            |
   |__________________________|
```

This complete our work of translating *NSpec* $\sqsubseteq_X$ *Impl* to *Prog* $\models_X$ *Unreachability*(*error*). Based on this translation, the Theorem 4 in [13] can be transferred to our CSP with arrays.

Theorem 5.22. *For any NSpec translatable to a* $SLTS_N$ *by our previous procedure, trace refinement checking on it is decidable.*

But a more interesting result can only be a conjecture.

Conjecture 5.2. *For any NSpec that is DI-explicit, trace refinement checking on it is decidable.*

Since our translation also decides on DI-explicitness of specifications, a by-product of our results is that it extends one of the theorem in [4] on trace refinement monotonicity.

6 Discussion

Our translation procedure depends crucially on the algorithm of translating *NSpec* to a Unity acceptor, which is the bottleneck for the whole work. But the algorithm is a compromise between termination and accuracy. We cannot do an exact symbolic execution of *NSpec* to eliminate all false negatives for it can easily lead to infinite symbolic states. Approximation is the best we can hope for in practice if we want a terminating algorithm.

However, with some careful techniques there are still spaces to improve our result and reduce the chances of false negatives.

Firstly we can notice that false negatives can be generated at two stages of our algorithm: either at the first stage of checking **C1** and **C2** or at second stage of checking **C3**. False negatives could be reduced if we could improve the chance of *NSpec* passing the two checks.

We have already see that introducing new array variables can help us get *NSpec* passing the first check (or more accurately **C2**). We could also see that if a new element could be added to the symbolic data states to bookkeep the number of different DI values we have collected so far from communication, we can improve the chance of *NSpec* passing the second check. This number helps save some information that would otherwise be lost when reducing recorded co-variables.

Similarly, even more improvement can be made if we give up our compositional translation approach and take into account the global behaviour conjunction effect of alphabetized parallel.

More generally we can broaden the scope of our results by removing the norm condition on *Spec*.

- It is not difficult to transform LLQs to satisfy the disjoint communication channel requirement in the norm condition. It enables us to reuse the algorithm above for the translation from LLQs to $SLTS_N$s.
- Other parallel operators like interface parallel can be handled by devising special algorithms to expand and compose several basic $SLTS_N$s into one larger one.
- Hiding on $SLTS_N$s can be done by renaming the symbolic events of hided transition to τ, but we needs some care here. We must make sure that a hided transition has a disjoint guard from its sibling transitions and it has no data components left unresolved; that is, the event can be reduced to a pure output event, so no new recorded variables or DI value inputs from environment are needed.

7 Conclusion and future work

Monotonicity is the major barrier to translating *Spec* $\sqsubseteq_X$ *Impl* to *Prog* $\models_X$ *Unreachability*(*error*). For the trace model, the problem is due to the specifications that, though syntactically data independent, could violate "data independence" in some semantical sense. By introducing the notion of DI-explicit specifications, we conjecture that all trace refinement checking on DI-explicit specifications is translatable to *Prog* $\models_X$ *Unreachability*(*error*) if necessary advanced data structures are available.

To that end, a formal translation procedure is devised to work on a fragment of finite-control CSP with boolean DI array, which can not only decide on the DI-explicitness of specifications, but also translate any refinement checking problem on it to some Unity unreachability problem. The crucial point of the translation happens in translating *Spec* satisfying the norm condition to a Unity acceptor, where the checking of DI-explicitness on *Spec* is also conducted on-the-fly. The algorithm relies on an over-approximate symbolic execution of *Spec*, so it may sometimes give false negatives on DI-explicitness.

This translation procedure can be improved in many ways to reduce false negative and broaden the applicability to *Spec* without the norm condition assumption. All these make the decidability results on Unity resettable DI arrays readily available to a large class of CSP trace refinement problem.

Obviously, it is only the first step in our efforts of relating our recent DI work in Unity-like languages to our previous DI work in CSP. It has solved the problem on fixed-finite DI arrays (, which can be encoded in boolean DI arrays) as it is. But it remains to be seen how DI arrays indexed by one DI type (e.g., X) and containing members of another (e.g., Y) can be bought into picture.

Furthermore, our results are largely limited to the trace model of CSP. But reachability/unreachability is a very general framework. With it, some more advanced model like stable failure are also likely to be translatable, even though the actual procedure may be more complex. DI-explicitness and nondeterministic variable reduction need be explored in stable failure model too; most of the techniques we developed for the trace model, like partitions and decorations, are expected to be applicable there. Indeed, it is the framework for translation and the techniques for analyzing DI boolean expression that, we think, comprise the most important contribution of the current work.

Another very interesting development can happen in threshold calculation [4]. Decidability is very important result, but only after the threshold be calculated can we bring verification problems directly to the raw power of FDR.

Acknowledgements

This work is funded by the EPSRC standard research grant 'Exploiting data independence', GR/M32900 and a grant from US ONR. The third author is also affiliated to the Mathematical Institute, Belgrade, and supported in part by a grant from the Intel Corporation.

References

[1] R. Alur and T. Henzinger. Computer-aided verification: An introduction to model building and model checking for concurrent systems. Draft, 1999.

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

[3] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. In *Symposium on Theoretical Aspects of Computer Science*, pages 421–433, 1989.

[4] R. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University Computing Laboratory, 1999.

[5] R. Lazić, T. Newcomb, and A. Roscoe. On model checking data-independent systems with arrays without reset. Technical Report RR-02-02, Oxford University Computing Laboratory, Jan. 2002. To appear in the Journal of Theory and Practice of Logic Programming.

[6] R. Lazić and D. Nowak. A unifying approach to data independence. In *Proceedings of the 11th International Conference on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 581–595. Springer-Verlag, Aug. 2000.

[7] R. Lazić and D. Nowak. On a semantic definition of data independence. In *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications*, 2003. To appear.

[8] R. S. Lazic and A. Roscoe. Data independence with generalised predicate symbols. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 319–325, Las Vegas, Nevada, USA, June 1999. CSREA.

[9] H. Lin. Symbolic transition graph with assignment. In *Proceedings of the 7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 1996.

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

[11] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[12] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[13] A. Roscoe and R. Lazić. 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, Sept. 2001.

[14] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 353–378. Prentice Hall, 1994.

[15] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[16] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science*, pages 332–344, Washington, DC, 1986.

[17] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193, 1986.