



Combining the Box Structure Development Method and CSP for Software Development

Philippa J. Hopcroft¹

*Oxford University Computing Laboratory
United Kingdom*

Guy H. Broadfoot²

*Verum Consultants
The Netherlands*

Abstract

In this paper, we combine the Box Structure Development Method (BSDM) [6,8] and CSP [3,10], integrating them into industrial software development processes. BSDM was developed with practical software projects in mind and provides a framework for developing formal design specifications that are fully traceable to the informal requirements. It integrates well into an industrial setting and forms an ideal bridge between the actual system being developed and the abstract models used for formal analysis. CSP complements BSDM by providing the mathematical framework for formal verification, together with its model checker FDR. In this paper, we present generic algorithms for translating specifications from BSDM into CSP, illustrate how they can be formally verified using FDR and summarise an industrial case-study.

Keywords: Software design verification, CSP, FDR, Box Structure Development Method.

1 Introduction

In this paper, we combine two existing and complementary formal methods, integrating them into industrial software development processes. The first of these is the Box Structure Development Method (BSDM), originally developed by Mills [6], and later extended by others (for example, see [8]). The second

¹ Email: philippa.hopcroft@comlab.ox.ac.uk

² Email: guy.broadfoot@verum.com

approach is the process algebra CSP [3,10], together with its model checker FDR [9,2].

The problem domain motivating this work is the development of practical software systems where the complexity and concurrency make them increasingly unreliable with conventional testing methods alone. Many formal methods have been developed over the years, for example [3,4,1]. However, integrating them into industrial software development environments has proved to be difficult. There are many reasons cited ranging from scalability to educational issues, depending on the formal approach in question. A common problem we have encountered in practice is that the formal specifications required for formal verification are typically inaccessible to domain experts and business analysts who have the necessary product knowledge. Training them in the formal language at the start of every project is infeasible. This leads to the very people who are essential to the validation process being excluded from it.

BSDM was developed with practical software projects in mind and provides an ideal vehicle for bridging the gap between the abstract formal models and the software development process in practice. It uses a requirements analysis technique that leads to a formal specification traceable to the original informal specification. This specification is then transformed into an implementation via a number of refinement steps. Together with the sequence-based specification method [7], from the second author's experience in industry, this approach integrates well into a practical software development process of CMM level 2 and above. CSP complements BSDM by providing the essential mathematical framework for formal verification, together with the model checker FDR which provides the necessary automation.

The paper is organised as follows. In Section 2, we give an overview of the two formal frameworks in question. Sections 3 and 4 describe how BSDM's functional views can be modelled in CSP, together with generic algorithms for constructing them. The scope for automated analysis using the FDR model checker is discussed in Section 5. In Section 6, we give a brief overview of how our approach can be extended to model some of the abstraction techniques used in BSDM. Finally, we present our conclusions, an industrial application and future work in Section 7.

2 Background

2.1 Box structure development method

The box structure development method [5,6] defines three functional views of a system, forming an abstraction hierarchy that allows for stepwise refinement

and verification: A *black box* is a state-free description of the external view of the system; a *state box* is derived from the black box and introduces internal state; and the *clear box* view is an implementation of the state box. Each view must be complete, consistent and traceably correct.

In practice, we use this methodology for developing rigorous design specifications of the functional behaviour of all the system components from their requirements, and therefore currently limit ourselves to the black box and state box views only. Extending this to include clear boxes, or other approaches for generating code from the state box descriptions, is part of future work.

Black boxes A black box specification is the most abstract view and defines the external behaviour of a system or component in terms of input stimuli from and output responses observable to its environment; S and R denote the set of stimuli and responses respectively. A black box is characterised by a total black box function $BB : S^* \rightarrow R$ that maps stimulus history to responses, where S^* is the set of all finite sequences over S .

In practice, there may be a stimulus that does not invoke a response; for completeness, a special response *null* is added to the set R to handle this case. For example, in the initial state of a system prior to any stimulus input, no response can (spontaneously) occur. Sequences of stimuli that are illegal must be included and are mapped to the special value $\omega \in R$. Illegal sequences are extension-closed: all extensions of illegal sequences are also illegal.

The black box is developed from the informal requirements and is required to be complete, consistent and traceably correct with respect to them. In practice, this is difficult to achieve, since requirements are often incomplete, inconsistent and can run into the hundreds of pages. In [7], Prowell and Poore present the *sequence-based software specification* method for systematically defining a black box specification that ensures completeness, consistency and traceability with the original requirements. This method involves the ordered enumeration of all possible stimuli sequences in S^* in order of length, starting with the empty sequence, and the assignment of a response from R to each one. For each length, the ordering must be complete. During this enumeration, equivalences between sequences may arise: Two sequences $u, v \in S^*$ are *Mealy* equivalent, written $u \equiv_{\rho_{Me}} v$, precisely when all nonempty extensions of them result in the same response.

When such an equivalence arises between a sequence u and a previously enumerated one v , the response associated with u is noted, together with its equivalence to v . If no such equivalences exist, then u is said to be irreducible. The set S^* is thus partitioned into a finite set of equivalence classes as defined by ρ_{Me} where, for every sequence $u \in S^*$, there is a unique irreducible normal

form v such that $u \in [v]_{\rho_{Me}}$; these are known as *canonical sequences*.

Using the format in [8], the result of this sequence enumeration is a set of enumeration tables, one for each canonical sequence c with the following form:

Canonical sequence : c			
<i>Stimulus</i>	<i>Response</i>	<i>Equiv</i>	<i>Trace rule</i>

Traceability to the informal requirements is achieved by insisting that every design decision, regarding the assignment of responses and equivalences found, is justified by explicit reference to the corresponding informal requirements (captured in the column *Trace rule*). During this process, it frequently occurs that choices are required concerning issues about which the informal specification is silent, ambiguous or inconsistent. In these cases, new requirements must be formulated jointly with the domain experts. When applied within a well organised software engineering process (for example, of CMM level 2 and above), this approach is very effective at eliminating ambiguities and omissions in the requirements and retaining the involvement of domain experts throughout the design stage.

The enumeration process is complete once all sequences have been assigned to an equivalence class. Example 3.3 illustrates a complete enumeration in the tabular form. Enumeration specifications that are strongly bisimilar are regarded as equivalent (since they encode the same black box).

State boxes A state box specification is the next step towards implementation. It is derived from a black box and they are expected to be behaviourally equivalent. A state box introduces state variables to capture distinctive characteristics of the stimulus history, such that its behaviour is equivalent to that of the black box. It is characterised by a total function $SB : (Q \times S) \rightarrow (Q \times R)$, where Q , S and R are the sets of states, stimuli and responses respectively.

The states in Q represent characteristic predicates, one distinct state for each equivalence class, over the partition of S^* . Defining state variables and their unique instantiations for each equivalence class is achieved through a process called the *canonical sequence analysis* and is described in [8]. Informally, this process involves inventing variables that capture the conditions of every stimuli sequence. A suitable range of values for each variable is defined, such that the combination of variable values is unique for every canonical sequence. The introduction of state variables removes the need to refer to stimulus history and provides a more intuitive description for implementation purposes. Once such variables have been defined, the construction of the state boxes is

straightforward. However, in practice, defining suitable variables (within the context of the overall design) can be difficult and lead to errors being injected at this stage.

A black box and state box with functions $BB : S^* \rightarrow R$ and $SB : (Q \times S) \rightarrow (Q \times R)$ respectively are behaviourally equivalent precisely when, for every sequence in S^* , they invoke the same response. The state box specification is represented as a set of tables, one for each stimulus $s \in S$ with the following form:

Stimulus: s			
Current state	Response	New state	Trace rule

Example 4.3 illustrates this notation. In a complete, consistent state box, one can determine the unique response given for any stimulus $s \in S$ in any given state.

2.2 CSP and the FDR model checker

CSP [3] is a process algebra for describing concurrent processes that communicate with one another or their environment. It has a rich language and a collection of semantic models for reasoning about process behaviour. We give a brief overview here; see [10] for further details.

A process is defined in terms of synchronous, atomic *events* that it can perform with its environment: $a \rightarrow P$ can initially perform event a and then act like process P . We write $?x : A \rightarrow P(x)$ (prefix choice construct) to denote the process that is willing to perform any event in A and then behave like process $P(x)$. Channels carry sets of events: $c?x \rightarrow P_x$ inputs a value x from channel c and then acts like P_x .

CSP has the following choice operators: $P \square Q$ denotes an external choice between P and Q ; the initial events of both processes are offered to the environment. $P \sqcap Q$ denotes an internal choice between P and Q ; the process can act like either P or Q , with the choice being made nondeterministically. Processes can be placed in parallel, where they synchronise upon (all or specified) common events or interleaved, where they run independently of one another.

For the purposes of this paper, it suffices to introduce the simplest semantic model in CSP, known as the *traces* model. A *trace* is a sequence of events (visible to the environment). For any process P , $traces(P)$ is the set of all finite traces of P ; furthermore, $traces(P)$ is non-empty (always contains empty trace) and prefix closed. A process Q *trace-refines* a process P , written $P \sqsubseteq_T Q$, precisely when every trace of Q is also a trace of P . The *traces* model

allows us to verify safety properties only. Richer semantic models exist in CSP to capture a broader scope of properties such as nondeterminism and other liveness conditions. We refer to these models in Section 6 when extending our approach to handle underspecified specifications.

FDR [9,2] is a mature model checker for CSP that provides fully automated verification of refinement (for all semantic models), determinism, deadlock freedom and livelock freedom, together with extensive debugging facilities.

3 Modelling black boxes in CSP

In this section, we show how black boxes can be modelled in CSP and present a generic algorithm for their construction. The sequence-based specification of the black box describes a finite-state automaton (in this case, a Mealy machine), where the states represent the equivalence classes and the arcs capture the corresponding stimulus-response pairs between them. Such systems are intuitively modelled in CSP as a set of mutually recursive processes, one representing each equivalence class, and defined with stimulus-response pairs of events leading to the process that represents the corresponding equivalence state.

We start by defining what it means for a CSP process to model a black box specification. For a black box function $BB : S^* \rightarrow R$, we will define a CSP process P with stimulus and response events from S and R respectively; thus the alphabet of P will be $S \cup R$. For some trace tr , we write $tr \upharpoonright S$ to denote the trace whose members are those of tr that are also in set S . For example, the trace $tr = tr' \frown \langle r \rangle$ for some trace tr' and response r , represents the stimulus sequence $tr' \upharpoonright S$ with response r .

Definition 3.1 A CSP process P is said to model a black box function $BB : S^* \rightarrow R$, precisely when the following conditions are satisfied:

- (i) P must not introduce new behaviour (not in BB):

$$\forall tr, tr' \in \text{traces}(P); r \in R \bullet tr = tr' \frown \langle r \rangle \Rightarrow BB(tr \upharpoonright S) = r.$$

- (ii) P must model the whole domain of BB and map each one accordingly:

$$\forall z \in S^+; r \in R \bullet$$

$$BB(z) = r \Rightarrow (\exists tr, tr' \in \text{traces}(P) \bullet tr = tr' \frown \langle r \rangle \wedge tr \upharpoonright S = z).$$

where $S^+ = S^* - \{\langle \rangle\}$. The case $z = \langle \rangle$ is modelled by the empty trace $\langle \rangle$ which, by definition of the traces-model, is a member of $\text{traces}(P)$.

- (iii) P reflects the stimulus-response pairing pattern. Since $\text{traces}(P)$ is prefix-closed, we specify this as:

$$\forall tr : \text{traces}(P) \bullet \#(tr \upharpoonright R) \leq \#(tr \upharpoonright S) \leq \#(tr \upharpoonright R) + 1.$$

where $\#(tr \upharpoonright X)$ returns the number of times an event in X occurs in trace tr .

(iv) P is a deterministic process, reflecting the fact that BB is a function.

Condition 3 is a characteristic of the black box specification that must be observed by the CSP model. In terms of the corresponding finite state automaton, from any given state, actions only of the form of a stimulus followed by a response can be performed. Without this property in our definition, one could construct a CSP process P such that $\langle s, r, r \rangle \in \text{traces}(P)$. Given that $BB(\langle s \rangle) = r$, this trace would satisfy the first two conditions above. However, this does not accurately model the pairing assumption made by the black box specification of the system.

In the case where a single stimulus does invoke a sequence of consecutive responses, one would still capture this as a single abstract response to model the fact that they are treated atomically and are a result of a single stimulus input.

Definition 3.2 A CSP process satisfying Definition 3.1 and a sequence-based specification are said to be *congruent* precisely when they encode the same black box function.

For the purposes of clarity, we capture an enumeration table for some canonical sequence c_i as the set $\text{Rows}_{BB}(c_i)$ comprising one element per row of the table and taking the form (s, r, c_j) , where s , r and c_j represent the stimulus, response and equivalence columns respectively. For the purposes of the CSP model, we ignore the fourth column capturing the traceability to informal requirements, as it is not relevant in the CSP analysis of the system.

Algorithm 1 For a given sequence-based specification that encodes a black box function $BB : S^* \rightarrow R$, a corresponding CSP process P_{BB} is defined as follows:

- (i) For every canonical sequence c_i in the enumeration, a CSP process P_{c_i} is defined as follows:

$$P_{c_i} = \Box \{ Q(s, r, c_j) \mid (s, r, c_j) \in \text{Rows}_{BB}(c_i) \}$$

where

$$Q(s, r, c_j) = s \rightarrow r \rightarrow P_{c_j}$$

This produces a collection of mutually recursive CSP processes P_{c_0}, \dots, P_{c_n} , one process P_{c_i} for each canonical sequence c_i in the enumeration.

- (ii) The CSP process P_{BB} is then defined as $P_{BB} = P_{c_0}$, where c_0 represents the shortest canonical sequence, namely $\langle \rangle$.

Due to space restrictions, we use a very simple example below to illustrate this algorithm. However, we discuss our industrial experiences in Section 7.

Example 3.3 Consider a highly simplified vending machine with the following sets of stimuli and responses:

$$S = \{coin, tea, coffee, cancel, accept\}$$

$$R = \{return_coin, dispense_tea, dispense_coffee, menu, pay_coin, confirm, null\}$$

Assume there are four canonical sequences enumerated as follows:

$c_0 : \langle \rangle$			$c_1 : \langle coin \rangle$		
Stimulus	Response	Equiv	Stimulus	Response	Equiv
coin	menu	c_1	coin	return_coin	c_1
tea	pay_coin	c_0	tea	confirm	c_2
coffee	pay_coin	c_0	coffee	confirm	c_3
cancel	null	c_0	cancel	return_coin	c_0
accept	null	c_0	accept	null	c_1

$c_2 : \langle coin, tea \rangle$			$c_3 : \langle coin, coffee \rangle$		
Stimulus	Response	Equiv	Stimulus	Response	Equiv
coin	return_coin	c_2	coin	return_coin	c_3
tea	null	c_2	tea	null	c_3
coffee	null	c_2	coffee	null	c_3
cancel	menu	c_1	cancel	menu	c_1
accept	dispense_tea	c_0	accept	dispense_coffee	c_0

This enumeration encodes a complete black box specification: for any sequence $z \in S^*$, one can derive the corresponding response directly from the tables. The *null* response is used to represent the lack of response from the system.

Using Algorithm 1, the CSP process generated is $P = P_{c_0}$ where:

$$P_{c_0} = coin \rightarrow menu \rightarrow P_{c_1}$$

$$\square \quad tea \rightarrow pay_coin \rightarrow P_{c_0}$$

$$\square \quad coffee \rightarrow pay_coin \rightarrow P_{c_0}$$

$$\square \quad cancel \rightarrow null \rightarrow P_{c_0}$$

$$\square \quad accept \rightarrow null \rightarrow P_{c_0}$$

Processes P_{c_1} , P_{c_2} and P_{c_3} are defined accordingly from the tables. One can step through the traces of P and see how each one directly corresponds to the stimulus sequence-response mapping in the enumeration.

Proposition 3.4 *For a given sequence-based specification that encodes a black box function BB , if a CSP process P is constructed according to Algorithm 1, then P models the black box function BB according to Definition 3.1 and they are therefore congruent.*

Proof straightforward and omitted due to lack of space.

The resulting CSP processes are deterministic, reflecting the fact that the black box specification is indeed a function. For specifying the design of a component, such deterministic models suffice, since one does not choose to implement nondeterministic software. The purpose of these formal design specifications is to ensure that they are complete and unambiguous when passed on for implementation.

However, the ability to abstract away design details is an important requirement in practice. For example, when specifying interfaces of components and verifying their interactions, it is essential to be able to hide internal actions. This naturally leads to nondeterminism. The CSP framework captures nondeterminism very effectively and the sequence-based specification method provides a practical method for under-specifying components for these purposes, without conflicting the functional foundations of their approach. In Section 6, we discuss how our approach can be extended to handle nondeterminism.

4 Modelling state boxes in CSP

The next step towards implementation is deriving a behaviourally equivalent state box from the black box. A state box defines a total function $SB : (Q \times S) \rightarrow (Q \times R)$, where Q is the set of states, S is the set of stimuli and R is the set of responses. Each state $q \in Q$ represents a characteristic predicate χ_q for an equivalence class defined by ρ_{Me} over S^* ; we write $[q]_{\rho_{Me}}$ to denote the set of sequences in S^* for which χ_q holds true.

In this section, we illustrate how state boxes can be modelled in CSP and present a generic algorithm for constructing them. We start by defining the traces properties to be satisfied by a CSP model for a state box function.

Definition 4.1 A CSP process P is said to model a state box function $SB : (Q \times S) \rightarrow (Q \times R)$, precisely when the following conditions are satisfied:

(i) P must not introduce new behaviour (not in SB):

$$\begin{aligned} &\forall tr, tr' \in traces(P); s \in S; r \in R \bullet tr = tr' \frown \langle s, r \rangle \Rightarrow \\ &\quad \exists q, q' \in Q \bullet SB(q, s) = (q', r) \wedge tr' \upharpoonright S \in [q]_{\rho_{Me}} \wedge tr \upharpoonright S \in [q']_{\rho_{Me}}. \end{aligned}$$

(ii) Every mapping in SB is modelled by P :

$$\begin{aligned} &\forall q, q' \in Q; s \in S; r \in R \bullet SB(q, s) = (q', r) \Rightarrow \\ &\quad (\forall tr \in traces(P) \bullet tr \upharpoonright S \in [q]_{\rho_{Me}} \Rightarrow \\ &\quad \quad tr \frown \langle s, r \rangle \in traces(P) \wedge (tr \upharpoonright S) \frown \langle s \rangle \in [q']_{\rho_{Me}}). \end{aligned}$$

By induction, this implies that all mappings in SB are modelled by P (since $traces(P)$ is always non-empty, by definition of the traces-model).

(iii) P satisfies the stimulus-response pairing pattern:

$$\forall tr : traces(P) \bullet \#(tr \upharpoonright R) \leq \#(tr \upharpoonright S) \leq \#(tr \upharpoonright R) + 1.$$

(iv) P is a deterministic process, reflecting the fact that SB is a function.

The traces of the CSP process are composed of stimuli and responses in S and R respectively and clearly make no reference to the state data. This allows for a direct comparison between the traces of the process modelling the black box and that of the state box. Any errors introduced by design decisions made from the sequence-based enumeration of the black box to the state box in terms of the system's behaviour are easily found in FDR by verifying whether the corresponding CSP processes are traces-equivalent.

Definition 4.2 A CSP process satisfying Definition 4.1 and a state box are said to be *congruent* precisely when they model the same state box function.

We present a generic algorithm below for translating state box specifications into equivalent CSP models. We capture the information presented in a state box for a stimulus $s \in S$ as the set $Rows_{SB}(s)$ comprising one element per row and taking the form (q, r, q') , where q is the current state, r is the response invoked by s in state q and q' is the updated state.

Algorithm 2 Given a state box specification with function $SB : (Q \times S) \rightarrow (Q \times R)$, a corresponding CSP process P_{SB} is defined as follows:

(i) Let x_1, \dots, x_k denote the state variables as defined by the state box.

(ii) The process P_{SB} is defined as follows:

$$P_{SB} = P'_{SB}(init_1, \dots, init_k)$$

where $init_1, \dots, init_k$ are the initial values of the state variables x_1, \dots, x_k , as defined in the state box.

(iii) The process P'_{SB} is defined as follows:

$$P'_{SB}(x_1, \dots, x_k) = \Box \{ Q(s) \mid s \in S \}$$

There is one $Q(s)$ per state box for stimulus s .

(iv) For each stimulus $s \in S$, $Q(s)$ is defined as follows:

$$Q(s) = \Box \{ Q'(q, s, r, q') \mid (q, r, q') \in Rows_{SB}(s) \}$$

where

$$Q'(q, s, r, q') = q \ \& \ s \rightarrow r \rightarrow P'_{SB}(q')$$

For a stimulus s , there is one process $Q'(q, s, r, q')$ for each row in its state box. The value of q reflects the precondition for which s invokes

response r and state update q' , and is modelled as a boolean guard in CSP.

Example 4.3 Recall the simple vending machine presented in Example 3.3. Consider the following state box specification derived from that sequence-based specification, where the characteristic predicates for each equivalence class (each denoted by a canonical sequence) is defined by two state variables *Coin* and *Drink* as follows:

Equivalence class	Characteristic predicate
$[\langle \rangle]_{\rho_{Me}}$	$Coin = false$
$[\langle coin \rangle]_{\rho_{Me}}$	$Coin = true \wedge Drink = none$
$[\langle coin, tea \rangle]_{\rho_{Me}}$	$Coin = true \wedge Drink = tea$
$[\langle coin, coffee \rangle]_{\rho_{Me}}$	$Coin = true \wedge Drink = coffee$

where *Coin* and *Drink* are defined with the range of values $\{true, false\}$ and $\{none, tea, coffee\}$ respectively. The state box tables, one for each stimulus, are derived from the enumeration tables in Example 3.3 and the characteristic predicates defined above. For example, (ignoring the 4th column *Trace rule*) the state boxes for stimuli *tea* and *cancel* are defined as:

Stimulus: tea		
Current state	Response	New state
$Coin = false$	<i>pay_coin</i>	<i>No update</i>
$Coin = true \wedge Drink = none$	<i>confirm</i>	$Drink = tea$
$Coin = true \wedge Drink \neq none$	<i>null</i>	<i>No update</i>

Stimulus: cancel		
Current state	Response	New state
$Coin = false$	<i>null</i>	<i>No update</i>
$Coin = true \wedge Drink = none$	<i>return_coin</i>	$Coin = false$
$Coin = true \wedge Drink \neq none$	<i>menu</i>	$Drink = none$

Assuming the initial values for *Coin* and *Drink* are *false* and *none* respectively, the CSP process is defined as $P_{SB} = P(false, none)$ where:

$$\begin{aligned}
P(c, d) = & \\
& \neg c \ \& \ coin \rightarrow menu \rightarrow P(true, d) \\
& \square \ c \ \& \ coin \rightarrow return_coin \rightarrow P(c, d) \\
& \square \ \neg c \ \& \ ?x : \{tea, coffee\} \rightarrow pay_coin \rightarrow P(c, d) \\
& \square \ c \wedge d = none \ \& \ ?x : \{tea, coffee\} \rightarrow confirm \rightarrow P(c, x) \\
& \square \ c \wedge d \neq none \ \& \ ?x : \{tea, coffee\} \rightarrow null \rightarrow P(c, d) \\
& \square \ \neg c \ \& \ cancel \rightarrow null \rightarrow P(c, d) \\
& \square \ c \wedge d = none \ \& \ cancel \rightarrow return_coin \rightarrow P(false, d)
\end{aligned}$$

- $c \wedge d \neq \text{none} \ \& \ \text{cancel} \rightarrow \text{menu} \rightarrow P(c, \text{none})$
- $\neg c \vee d = \text{none} \ \& \ \text{accept} \rightarrow \text{null} \rightarrow P(c, d)$
- $c \wedge d \neq \text{none} \ \& \ \text{accept} \rightarrow \text{dispense} . d \rightarrow P(\text{false}, \text{none})$

Each guarded choice is directly derived from the state box tables. For example, choices 3, 4 and 5 are derived from the tables for *tea* and *coffee* respectively.

Proposition 4.4 *For a given state box that encodes a complete, consistent state box function SB , if a CSP process P is constructed according to Algorithm 2, then P models SB according to Definition 4.1 and they are therefore congruent.*

Proof straightforward and omitted due to lack of space.

5 Automated analysis using FDR

In this section, we give a brief summary of the analysis we do in FDR using this framework in industry.

Verifying black boxes and state boxes At each stage of the box structure development of a design, once the specification is modelled in CSP, properties such as completeness, determinism and control laws specified in the system’s requirements can be formulated and verified using FDR.

Another requirement to be verified in the state boxes is that the characteristic predicates defined as state data are non-overlapping and complete. In practice, errors during this phase are frequently introduced. Both properties are easily checked in FDR: The first of these is captured by checking for non-determinism; if there are overlapping predicates, then there will be a state where, for a given stimulus s , at least two guards evaluate to true, leading to a nondeterministic choice as to which of these is performed. If we want to distinguish between cases where the resulting response and state in both cases are the same, then additional events can be added to the model automatically to ensure that this error is still captured as nondeterministic behaviour. The second property can be formulated as a liveness condition and also verified using FDR.

Verifying behavioural equivalence Every step in the box structure development must be verified. Regarding black boxes and state boxes, one must verify that they are behaviourally equivalent (as defined in Section 2.1). Typically, this verification is done by deriving the black box behaviour from the state box and checking whether it is equivalent to the original black box specification. This is time consuming and infeasible in practice. The algorithms presented here allow both specifications to be translated into corresponding

CSP models respectively and the necessary verification checks to be done automatically in FDR.

Using Definitions 3.1 and 4.1, the behavioural equivalence between a sequence-based specification and a state box specification as defined above can be formulated in terms of the traces of the corresponding CSP processes: A deterministic sequence-based specification $Spec_{BB}$ and a deterministic state box specification $Spec_{SB}$ are behaviourally equivalent precisely when their respective CSP models P_{BB} and P_{SB} are traces-equivalent. This is automatically checked using FDR. If the check fails, then FDR produces counter-examples; due to the close correlation between the CSP traces and the tabular representation within BSDM, the counter-examples are simple to understand in either framework.

Verifying parallel composition The box structure method is component-centric in the development of designs. We typically use it to develop design and interface specifications for components, once the software architect has completed the overall structure and decomposed it into individual components. By translating them into CSP, we are able to place them in parallel according to the proposed architecture and formally verify the system as a whole. Properties such as deadlock, whether components invoke illegal behaviour in each other and control laws are formulated in CSP and automatically verified in FDR. The compositionality property of CSP provides the scalability for such development in practice.

6 Handling abstractions

In [7], a number of effective abstraction techniques are presented for avoiding over-specification in the sequence-based specification method (used for constructing the black boxes). We briefly summarise how our approach can be extended to handle two of these we use in industry.

To avoid over-specification within the sequence enumeration, details can be abstracted away in numerous ways, for example, by introducing abstract stimuli (rather than specifying the complete enumeration with actual concrete stimuli of the system). One must then show that the two representations are behaviourally equivalent. Such abstraction typically involves mapping sequences from distinct equivalence classes into a single sequence and defining predicates over them to capture their distinctive characteristics. These predicates can then be defined when that level of detail is required. For example, consider a vending machine where dispensing a drink involves checking a number of resources for availability. Enumerating this and identifying distinct equivalence classes for every combination would be inefficient.

An alternative approach is to abstract away this level of detail and introduce a predicate p that determines whether or not there are sufficient resources to enable a given drink d to be dispensed. Such predicates are typically defined in terms of prefix recursive functions over stimuli sequences. In this case, p may be defined as $Resource(h) \neq \emptyset$, where $Resource(h)$ is defined over stimuli sequences h and returns the set of available resources. A stimulus s whose response depends on this predicate (for example, *tea*), is then modelled by the special abstract stimuli $s : p$ and $s : \neg p$. For example, an enumeration table for a given canonical sequence c may be defined using these abstract stimuli as follows:

Stimulus	Response	Equiv	Trace rule
$s : Resource(h) \neq \emptyset$	<i>dispense</i> (s)	c_i	...
$s : Resource(h) = \emptyset$	<i>unavailable</i> (s)	c_j	...

where *Resource* must be defined for all stimuli sequences in the equivalence class characterised by the canonical sequence c and h represents the stimuli sequence history preceding all stimuli listed in the enumeration table.

Completeness and consistency is achieved by ensuring that the predicate and its negation are defined over the same sequence. See [7] for further details.

Algorithm 1 can be extended to handle predicates that are defined over stimulus history, by introducing state variables to capture the essential information (in the same way as is done for the state box specifications). Instead of referring to the sequence history, the predicates are computed in terms of the corresponding state data within the CSP models. Each stimulus s in an enumeration table of the form $s : p$ is modelled as a choice with the boolean guard p . For example, for the table above, we would have:

$$\begin{aligned}
 P(res) &= res \neq \emptyset \ \& \ s \rightarrow dispense.s \rightarrow P_{c_i}(f(res)) \\
 &\square \\
 &res = \emptyset \ \& \ s \rightarrow unavailable.s \rightarrow P_{c_j}(g(res))
 \end{aligned}$$

where *res* is the state variable and the functions f and g reflect state updates as defined by *Resource*.

Under-specification The sequence-based specification method uses predicates to enable the construction of under-specified black box specifications [7] (without breaking the function theory upon which this approach is built). This is an essential form of abstraction in practice, both for specifying component interfaces, where the internal actions are hidden from its environment, and for specifying requirements against which the design will be verified. Under-specification at the black box level of abstraction is achieved by introducing

undefined predicates to capture the fact that for a given stimulus s , there are a number of possible responses and subsequent behaviour, the decision of which is as yet undefined.

In ongoing research, we are looking at weakening Definition 3.1 and extending Algorithm 1 to construct corresponding CSP models and capture the potential nondeterminism. For example, for abstract stimuli of the form:

Stimulus	Response	Equiv	Trace rule
$s : p$	r_1	c_i	\dots
$s : \neg p$	r_2	c_j	\dots

where p is an undefined predicate, the CSP process is defined as follows:

$$P = (s \rightarrow r_1 \rightarrow P_{c_i}) \sqcap (s \rightarrow r_2 \rightarrow P_{c_j})$$

thereby potentially leading to nondeterminism after s is performed (depending on whether subsequent responses and behaviours are distinct).

In practice, we can specify component interfaces of a system using this approach and verify whether a proposed design (translated into CSP from BSDM) satisfies the intended interface automatically with FDR by using CSP refinement. Due to transitivity of refinement and monotonicity of the CSP operators with respect to refinement, CSP supports compositional development, enabling this approach to scale in practice. When modelling nondeterminism within the CSP models, the failures-model is used for all refinements.

7 Conclusion

In this paper, we gave an overview of how BSDM and CSP can be combined to reap the benefits of both in practice. We presented generic algorithms for converting black box and state box specifications into CSP and briefly discussed how this work could be extended to handle abstraction techniques such as underspecification. We use this approach to enhance the designs of software systems and verify them before the implementation phase starts; in practice, design errors are frequently the most expensive ones to resolve (in terms of resources and time) when only found during the testing phase.

There are a number of advantages of our proposed combined approach over some of the traditional formal methods, including CSP used on its own. The first one is its scope for integration into an organised industrial environment, achieved by BSDM and the sequence-based specification method. It is practical in the sense that it provides traceability between the formal specifications and the informal requirements, is accessible to software engineers and domain

experts without the need for extensive training, and from our experience of using this approach in industry over the last two years, it has shown to scale well (references to the experiences of others can be found in [8]). Secondly, the specifications developed using BSDM can be translated into other formal languages, in our case CSP, and it therefore acts as an effective bridge for introducing formal verification methods. Finally, the model checker FDR has provided the necessary automated tool support required in practice. This also includes effective and meaningful feedback regarding design errors found during the analysis. Due to the similarity between the tabular representation within BSDM and its corresponding description in terms of traces within CSP, provided one selects sensible naming conventions for each component's stimuli and responses within the system, it is very straightforward to interpret the counter-examples generated by FDR in either formalism.

Industrial experience We have used this combined approach for developing complex, event driven embedded control software controlling complex manufacturing machines. It has proven to be successful and scalable for the development of new components and the re-engineering of existing software components. For example, one of our applications involved the design and implementation of a process control Kernel for a machine based on 22 headless PC platforms running in parallel and communicating with each other via an internal Ethernet. The Kernel was designed as two concurrent processes, namely the Machine Controller and the State Controller processes, communicating via queues. The Machine Controller implemented the basic operational behaviour of the machine and controlled the other major subsystems. It was responsible for machine consistency and enforcing the control laws governing safe operation. It used interfaces provided by the subsystems it controlled and it implemented an interface used solely by the State Controller. The State Controller implemented the overall machine behaviour as seen by the GUI component and thus the machine operator.

The black box function of the Machine Controller design had 2,835 mappings in 47 equivalence classes. The length of the longest canonical sequence was 11 stimuli. The black box function of the State Controller design had 837 mappings and the length of the longest canonical sequence was 6 stimuli.

In addition to defining the design of the Kernel components as black boxes, the interfaces they implemented and those they used were defined as under-specified black boxes. Translating these specifications into CSP using the algorithms presented here enabled the following verifications to be performed automatically by FDR: (i) the designs of both Kernel processes were deterministic and implemented their respective interfaces correctly; (ii) the parallel composition of the State Controller and the process representing the interface

of the Machine Controller was deadlock free and behaved correctly according to the Machine Controller's interface; (iii) the parallel composition of the Machine Controller and the processes representing the interfaces of the other machine components was deadlock free and behaved correctly; (iv) the Machine Controller satisfied the control laws under all conditions.

The finished Kernel was approximately 20,000 executable lines of C++. The specification, design and implementation effort was approximately 16 man weeks. The final code was integrated on the target machine in one day. In the first 4 weeks of intensive use, 8 minor coding errors were found, none of which were specification or design errors; in more than 12 months since, no other errors have been detected. Rework to date is below 1%.

Future work We are currently formalising the work summarised in Section 6 and extending our approach with other practical abstraction techniques. Ongoing research is also being done regarding efficient modelling of constructs such as queues; we are seeking to classify common design patterns and process relationships, according to the modelling techniques appropriate to each. Other avenues of interest are: enabling the automated verification of different levels of abstraction in the enumeration specifications using FDR (for example, the abstract versus concrete stimuli); extending our approach for the refinement step from the state box to clear box in BSDM; and modelling real-time systems in this framework.

Acknowledgement

We thank Bill Roscoe and Michael Goldsmith for useful discussions and comments on this work. This research is funded by Verum Consultants, The Netherlands.

References

- [1] J. R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
- [2] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, 2003.
- [3] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [4] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [5] H. D. Mills. Stepwise refinement and verification in box structured systems. *Computer*, 21(6):23–26, 1988.
- [6] H. D. Mills, R. C. Linger, and A. R. Hevner. *Principles of Information Systems Analysis and Design*. Academic Press, 1986.

- [7] S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Trans. of Soft. Eng.*, 29(5):417–429, 2003.
- [8] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore. *Cleanroom Software Engineering - Technology and Process*. Addison-Wesley, 1998.
- [9] A. W. Roscoe. Model-checking CSP. In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 353–378. Prentice-Hall, 1994.
- [10] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.