Automatic generation of CSP || B skeletons from xUML models

Edward Turner, Helen Treharne, Steve Schneider [1], Neil Evans[2]

[1]Department of Computing, University of Surrey
[2]AWE plc Aldermaston

Abstract. CSP ∥ B is a formal approach to specification that combines CSP and B. In this paper we present our tool that automatically translates a subset of executable UML (xUML) models into CSP ∥ B, for the purpose of verification and increased validation at the early stages of a software engineering development lifecycle. The tool is being developed for our industrial collaborators, AWE plc, in order to strengthen their software engineering process which uses xUML. As part of this process, AWE and Kennedy Carter Ltd. have built an xUML to SPARK Ada code generator, which is also employed to contribute a higher level of safety assurance at the latter stages of the lifecycle. Our tool is based on a model-text transformation strategy that uses the xUML meta-model to map to CSP and B constructs. The tool generates machine readable CSP and B; we present a simple example to demonstrate the transformation strategy, and the analysis of the resulting specification.

1 Introduction

In this paper we discuss our approach to providing formal reasoning support for UML platform-independent models. The approach is being developed as part of a collaborative project with AWE plc. The application domain of interest is safety critical and therefore it is essential to achieve a high level of assurance in the safety of the models, i.e., they adhere to desirable behavioural properties and are deadlock-free. Current industrial practice involves validating UML models by examining and/or running numerous simulations. Our aim is to automatically generate CSP ∥ B [1] specifications, from executable UML (xUML) models [2], which can be formally analysed. The challenge is to identify an appropriate translation mapping with tool support so that a specifier's effort is spent on conducting formal analysis rather than on defining formal models.

The project will consider two different routes for developing a *CSP ∥ B specification generator*. Firstly, we will develop a specification generator using the xUML toolset provided by Kennedy Carter Ltd. (KC). The toolset offers the capability of code generation into C, C++ or Java from platform independent models. AWE have been working alongside Kennedy Carter to develop SPARK Ada translators from xUML. Thus, our tool will enable formal analysis support to fit into the AWE software development life cycle. Secondly, we will also investigate building a model generator using the Epsilon [3] toolset developed at York University. The contribution of this paper is a description of the first tool that we are currently building. The paper also demonstrates the need to develop an analysis framework so that added value can be gained from generating the formal CSP $\|$ B specification. The analysis framework is the way the specifications are verified for deadlock freedom and consistency. The process of generating the specifications themselves is also valuable because it forces us to think about ambiguities within an xUML model and resolves what are often implicit assumptions in xUML models.

We have chosen CSP $\|$ B as the underlying formalism because it provides a clean separation between control (in CSP [4]) and state (in B [5]). Moreover, its decompositional verification framework [1, 6] will be particularly relevant when it comes to analysing large xUML models. Much research exists on developing formal tool support for UML, including [7, 8]. Nonetheless, few focus on what else needs to be included in formalising large xUML models.

The paper begins with an overview of xUML and CSP $\|$ B. Section 2.1 illustrates a small running example. In Section 4 we present our tool which implements a model-text transformation strategy from xUML models to CSP $\|$ B specifications. It takes as input an xUML model written using the KC toolset, it is then invoked from within that toolset, and the output is a machine readable CSP $\|$ B specification. The strategy covers a large subset of concrete xUML and throughout the paper we identify restrictions on this subset. In Section 5 we discuss what analysis framework could be appropriate for the generated CSP $\|$ B specifications and Section 6 concludes with related work.

2 Executable UML

xUML is a coherent subset of UML 2.0 and supports six diagrams: Use Case, Domain Models, Sequence Diagrams, Class Diagrams, Collaboration Diagrams, and State Charts. In this paper we focus on class diagrams and state charts, since these are the main diagrams that are used when constructing xUML models using the Kennedy Carter toolset. We will refer to an xUML model comprising the class diagram and associated state charts as the *model* throughout the paper, and the corresponding CSP $\|$ B specification as the *specification*.

We also restrict ourselves to examining one domain within a model. Class diagrams enable the classes of a model to be defined together with the relationships between them. We currently support associations in xUML, which are binary by definition, but we do not handle generalisations. Class diagrams can have at most one state chart. The behaviour of instances (objects) of classes are described using state charts, each of which consists of a set of states and signals. Each state can define an entry action, whereas exit actions are not permitted. An action is defined by a sequence of statements described using the Action Specification Language (ASL) [9], which is a more low level language than OCL [10], and whose statements have no side effects and execute immediately. ASL statements can change the state of a system, be grouped into blocks, and execute concurrently. Hence, state actions can interleave their statements.

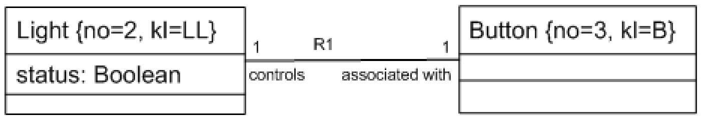

Fig. 1. Class diagram of the Lighting System

The ASL can be categorised into statements that perform object management, relationship management, state chart communication, sequential logic and assignments. The ASL we use in this paper is given as follows: `create` and `delete`, are used to create new instances and delete existing ones, respectively; `link` and `unlink`, are used to link and unlink instances over a specified association, respectively; `->` (pronounced '*navigate via*'), is used to retrieve instance handles over an association; `generate`, is used to send signals to a specified instance; and `<instance handle>.<attribute list> = <value list>` is used for writing to attributes of objects. We do not support ASL statements that deal with timers and access to methods from other domains.

Methods of class diagrams are also defined in terms of ASL. Currently, our work allows only for the definition of object scoped methods, and we must restrict the ASL they use, since methods will be mapped to B operations. For example, ASL for-loops are not permitted in methods and attribute values can only be written to once within an ASL block.

Traversing between states is achieved by processing signals on a queue, details of which are given in Section 4.3. A signal emerging from an action cannot be processed until the action has completed its execution (also known as *run to completion*). To ensure a complete description of behaviour for a state chart, xUML requires the specifier to define one of three effects of receiving any signal in any state: a signal can cause a *transition* between states and the execution of the entry action of the successor state; a signal can be *ignored*; or it may cause a run-time error (denoted a *cannot happen* effect).

2.1 Running Example

Consider an example lighting system consisting of two classes representing buttons and lights, shown in Figure 1. A button instance is paired with one light instance, and vice-versa, such that when a button is switched on, its attached light illuminates. Similarly, when a button is turned off, its light also turns off. Larger examples provided by KC have also been examined.

The only data we model explicitly is the boolean attribute, `status`, in the Light class, which denotes whether or not a light is illuminated. No object scoped methods are defined (however, our tool does support generating operations corresponding to object scoped methods and attribute accessor methods as shown in Section 4). The desired behaviour of button and light instances is captured by the state charts given in Figures 2 and 3, respectively.

We now describe the interaction of these state charts, so that lights can be turned on and off. Consider an instance of both classes in their `Off` states. To

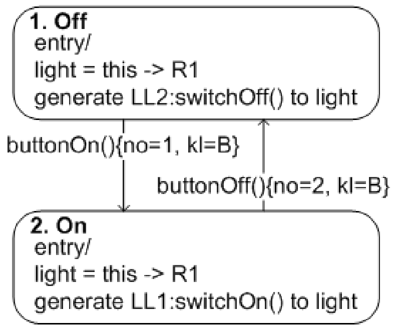

Fig. 2. State chart for Buttons

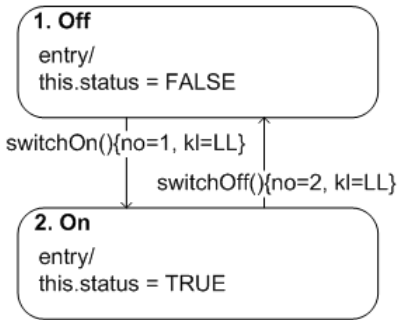

Fig. 3. State chart for Lights

State	buttonOn	buttonOff
Off	On	Ignore
On	Ignore	Off

Table 1. Effects of Button Signals

State	switchOn	switchOff
Off	On	Ignore
On	Ignore	Off

Table 2. Effects of Light Signals

turn on the light, we must send a `buttonOn` signal to the button. In our model, this task is neither the responsibility of buttons or lights; instead, it is externally generated. Receipt of this signal by the button triggers transition to its `On` state, and the execution of the defined entry action. Accordingly, we first obtain an instance handle to the attached light through the ASL navigation statement: `light = this -> R1`; where `this` refers to the instance handle of the button and `R1` is the name of the association being navigated. Subsequently, a `switchOn` signal is sent to the light, using the signal generation statement: `generate LL1:switchOn() to light`[1]. The attached light can process this signal and move to its own `On` state even if the button was still processing any remaining ASL statements (in this case there are none). The actions in this state then sets `status` to `TRUE`, to denote the light has been successfully illuminated. The procedure for turning off a light follows a similar pattern, but where the trigger is the `buttonOff` signal, which is also external.

Tables 1 and 2 show the effects of signals in particular states of our model. For example, we define in Table 1 that when a button is in the `Off` state and it processes a `buttonOff` signal, the signal should be ignored and discarded. Similarly, we ignore all unexpected signals.

Typical object scenarios are described in initialisation segments, e.g., to set up linked lights and buttons. Test methods can then be defined to generate signals to evolve system behaviour. Both these additions to an xUML model are achieved using ASL statements.

[1] Identifiers of signals and methods are made unique by the KC tool using automatically generated prefixes. Therefore, **switchOn** and **switchOff** signals are denoted by **LL1:switchOn()** and **LL2:switchOff()**.

3 CSP || B

CSP $\parallel$ B is an approach that combines an event-based notation with a state-based notation to facilitate the specification of systems with both complex flows of control and structured data. A degree of separation between the CSP [4] and B [5] aspects is maintained so that we can retain the use of existing tool support. The FDR model checker [11] and ProB [12] are both used to support the analysis and verification of CSP $\parallel$ B specifications.

CSP process expressions can be constructed from several operators, the ones used in this paper are: event sequencing ($\rightarrow$), (indexed) external choice ($\Box$), (indexed) interleaving ($|||$), (indexed) alphabetised parallel composition ($\parallel$), and interface parallel ($P \underset{S}{\parallel} Q$ where S is the synchronisation set). We also use conditional expressions (if then else) and local definitions (let within).

The failures of a process P consists of all (tr, X) where tr is a trace and X is a set of events P can refuse (see [13, 14]). A trace tr is said to be a trace of a process P if the process can perform the sequence of events in tr. The refinement relation in the failures model is denoted by $\sqsubseteq_F$. The refinement relation in the traces model is denoted by $\sqsubseteq_T$.

B specifications are structured using *machines*. Each machine contains some *state* describing the objects of interest, and *operations* to manipulate the state. A B machine also contains an *invariant* that declares properties of the state variables, and specifies what must be preserved by the execution of operations. Functions and relations are used to model complex state, e.g., R^{-1}, gives the relational inverse of a relation R and, given a set U, the relational image $R[U]$ is the set of objects (in the range of R) related to the elements of U.

A B operation takes the form **PRE** P **THEN** S **END** where P is a predicate and S represents the statements that update variables. In CSP $\parallel$ B we are particularly interested in operations without guards. The kind of B machines that we define are referred to as *non-blocking*. Hence, any deadlock in a CSP $\parallel$ B specification is as a result of the CSP processes deadlocking.

In CSP $\parallel$ B, the events of a CSP process trigger operation calls of a B machine, and the process is said to 'control' the B machine because its events cause state updates within the B machine via the operation calls. We refer to CSP processes as *controllers*. Structured events are used to pass values between the controller process and the B machine. For example, an event $e!x?y$, that outputs a value x and binds the variable y to an input value, corresponds to an operation call $y \longleftarrow e(x)$ that inputs x and outputs a value y. These events can contain a number of inputs/outputs or none. Our previous work has justified that it is meaningful to combine CSP processes and B machines [1]. We combine collections of controller/machine pairs using the architecture identified in [15] and shown in Figure 4. The association machine $ASSOC$ is controlled by the synchronising events of two processes P and Q. These processes define the collections of controllers representing the behaviour of instances of classes and the machines, M_1 and M_2, record and update the instances' attribute information. The $ASSOC$ machine tracks the associations that exist between instances.

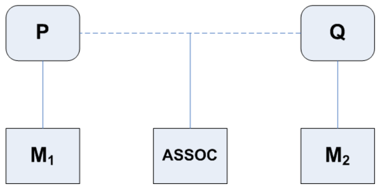

Fig. 4. Overview of CSP || B architecture

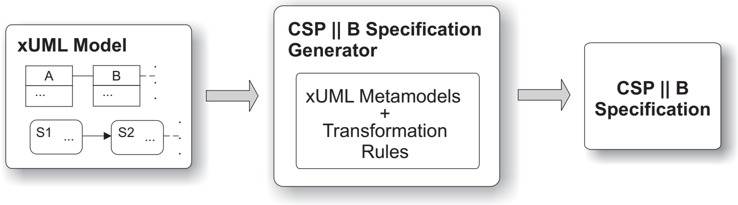
xUML Model
A
B
S1
S2
CSP || B Specification Generator
xUML Metamodels + Transformation Rules
CSP || B Specification

Fig. 5. The main components of our tool

4 Automatic Generator

The overall architecture of our tool is given in Figure 5. By importing an xUML model into our tool, we populate a suite of meta-models describing various aspects of the system. Our tool primarily relies on two of these; the representation of entities of the imported system, such as the classes and state charts used (part of the xUML meta-model), and the ASL used by system entities (stored within the ASL meta-model). The pattern for generating CSP and B then comprises a procedure that initiates a guided traversal of these meta-models, to access certain data, and the application of our transformation rules to obtain the elements of the formal specifications. The following sections discuss the steps for generating CSP and B from the meta-models.

4.1 Translating ASL to skeleton B machines

A B machine is created for each class in the class diagram of the xUML system. Figure 6 presents the B machine generated by our tool for the light class. A set, *llIH*, is used to specify the instance handles of the class, a subset of which, *llObj*, captures the current light instances in the system. The approach we use resembles the B style, presented in [7], for modelling objects and continues our previous work developed in [16]. Attributes are represented by functions, and simple operations are generated to query or modify their values. In addition, operations are provided to support the dynamic creation of objects. For example, **LL_create** creates a light instance and explicitly sets the *status* variable. In general, other methods of a class are transformed to skeleton B operations that specify the typing of input and output variables.

```
MACHINE
    LIGHT_MCH
SEES
    Bool_TYPE
SETS
    $llIH = \{ ll0 , ll1 \}$
VARIABLES
    $llObj , status$
INVARIANT
    $llObj \subseteq llIH \wedge$
    $status \in llObj \rightarrow BOOL$
INITIALISATION
    $llObj := \varnothing \;\|$
    $status := \varnothing$
OPERATIONS
 LL_create ( ih ) $\widehat{=}$
  PRE $ih \in llIH \wedge ih \notin llObj$  THEN
   $llObj := llObj \cup \{ ih \} \;\|$
   $status ( ih ) := false$
  END ;
vv $\longleftarrow$ LL_default_get_status ( ih ) $\widehat{=}$
  PRE $ih \in llObj$  THEN
   $vv := status ( ih )$
  END ;
 LL_default_set_status ( ih , val ) $\widehat{=}$
  PRE $ih \in llObj \wedge val \in BOOL$  THEN
   $status ( ih ) := val$
  END
END
```

Fig. 6. Skeleton machine for the Light class

```
link_R1 ( ll_ih , b_ih ) $\widehat{=}$
PRE $ll\_ih \in llObj \wedge b\_ih \in bObj$
  $ll\_ih \notin \mathsf{dom}\,(R1) \wedge b\_ih \notin \mathsf{ran}(R1)$
THEN
  $R1 := R1 \cup \{ ll\_ih \mapsto b\_ih \}$
END
```

```
ll_ih $\longleftarrow$ navigate_R1_from_B ( b_ih ) $\widehat{=}$
PRE $b\_ih \in bObj \wedge b\_ih \in \mathsf{ran}\,( R1 )$
THEN
  $ll\_ih := R1^{-1} ( b\_ih )$
END
```

Fig. 7. Example B operations for managing the $R1$ association

A single B machine is used to capture all the relationships used within an xUML model. For our example, a machine called *ASSOC* is generated, which defines the variable, $R1$, and its invariant representing the relationship between *llObj* and *bObj* (current button instances), as follows:

$$R1 \in llObj \leftrightarrow bObj$$

The relation can be specified in terms of the *llObj* and *bObj* variables since they are accessible via the **USES** structuring mechanism within the B-Method. The multiplicity constraints of $R1$ need to be maintained throughout the lifetime of the objects but the constraints are not discharged until we consider model consistency in Section 5.2. The *ASSOC* machine defines **link_R1** and **unlink_R1** operations linking/unlinking lights and buttons, and **navigate_R1_from_LL** and **navigate_R1_from_B** to traverse the association from either side, two of which are given in Figure 7.

4.2 Translating ASL to CSP

The general procedure for generating CSP for object behaviour is outlined in Algorithm 1. For each class c in an xUML system we define a controller process. Thus, for the button class, $\langle c \rangle\_CTRLS$ is instantiated as B_CTRLS, to represent the behaviour of all button instances, shown in Figure 8, where B denotes

Algorithm 1 An outline for generating CSP for object lifecycles

```
for all $c$ in classes of xUML model $m$ do
  if $c$ has a state chart then
    $\langle c\rangle\_SCTRL(ih) = \text{let}$ /* state chart behaviour for instance handle ih */
    for all states, $s$ in $sc$ do
      $\langle c\rangle\_\langle s\rangle\_\text{ENTRY} =$ sequence entry actions ending in$\langle c\rangle\_\langle s\rangle\_\text{STATE}$
      $\langle c\rangle\_\langle s\rangle\_\text{STATE} = \Box_{se \in \{signal\ effects\}}\ se \rightarrow \langle c\rangle\_\langle s\rangle\_STATE$
    end for
    within /* initial _STATE process*/
  end if
  if $c$ has a state chart then
    if $c$ has no attributes and no methods (excluding create/delete) then
      $\langle c\rangle\_CTRL(ih) = \langle c\rangle\_SCTRL(ih) \triangle (\langle c\rangle\_delete.ih \rightarrow STOP)$
    else
      $\langle c\rangle\_CTRL(ih) = ((\langle c\rangle\_SCTRL(ih) \triangle (\langle c\rangle\_delete!ih \rightarrow STOP)) \underset{\{\langle c\rangle\_delete.ih\}}{\|} \langle c\rangle\_DOPS(ih)$
    end if
  else
    if $c$ has no attributes and no methods (excluding create/delete) then
      $\langle c\rangle\_CTRL(ih) = \langle c\rangle\_delete!ih \rightarrow STOP$
    else
      $\langle c\rangle\_CTRL(ih) = \langle c\rangle\_DOPS(ih)$
    end if
  end if
  where $\langle c\rangle\_DOPS(ih) = (\Box_{o \in O}\ o \rightarrow \langle c\rangle\_DOPS(ih)) \Box (\langle c\rangle\_delete!ih \rightarrow STOP)$
  and $O$ is the set of methods and attribute accessor methods
  /* composition of all instances in $\langle c\rangle\_INSTANCES$, e.g., in simple case: */
  $\langle c\rangle\_CTRLS = |||_{ih \in \langle c\rangle\_INSTANCES}\ \langle c\rangle\_create!ih \rightarrow \langle c\rangle\_CTRL(ih)$
  $\alpha\langle c\rangle\_\text{CTRLS} =$ /* set of events in$\langle c\rangle\_$CTRLS */
end for
```

the class key letter and b denotes the instance handle to which it applies. This interleaving process means that all buttons instances act independently and only interact via the signal queues. In general, instances may be composed in parallel; the synchronising events would be those corresponding to creation/deletion methods, object scoped methods and attribute accessor methods. For all instances a create event initially occurs and the subsequent behaviour is defined using the B_CTRL parameterised process. For example, $B\_create!b1$, where $b1$ is a particular button instance, is a create event which triggers the corresponding B operation so that a new object is added, i.e., $b1$ is added to $bObj$. The creation of an object must make explicit all its initial attribute values and set the state machine into a particular state. This information is required by the corresponding B operation and the CSP *within* clause respectively. Currently, we do not support state charts containing an explicit initialisation state.

The B_CTRL process is responsible for describing the behaviour of the state chart associated with class B, defined in B_SCTRL. The B_CTRL process also

$$
\begin{array}{l}
B\_SCTRL(b) = \text{let} \\
\quad B\_Off\_ENTRY = navigate\_R1\_from\_B!b?ih \rightarrow \\
\qquad\qquad\qquad generate!ih!b!LL2\_switchOffSignal \rightarrow B\_Off\_STATE \\
\quad B\_Off\_STATE = \\
\qquad remove!b?\_!B1\_buttonOnSignal \rightarrow B\_On\_ENTRY \\
\qquad \Box \text{ /* branch representing ignored signals */} \\
\quad B\_On\_ENTRY = navigate\_R1\_from\_B!b?ih \rightarrow \\
\qquad\qquad\qquad generate!ih!b!LL1\_switchOnSignal \rightarrow B\_On\_STATE \\
\quad B\_On\_STATE = \\
\qquad remove!b?\_!B2\_buttonOffSignal \rightarrow B\_Off\_ENTRY \\
\qquad \Box \text{ /* branch representing ignored signals */} \\
\text{within } B\_Off\_STATE \\
B\_CTRL(b) = B\_SCTRL(b) \triangle (B\_delete!b \rightarrow STOP) \\
B\_CTRLS = \left|\left|\right|\right|_{b \in B\_INSTANCES} B\_create!b \rightarrow B\_CTRL(b)
\end{array}
$$

Fig. 8. Generated CSP for the Button state chart

ensures that when a deletion event is performed the state chart behaviour terminates. In general, any object scoped methods and attribute accessor methods of an object must be offered at all times while that object exists. An instantiation of process $\langle c \rangle\_DOPS$ would provide this behaviour.

The pattern for the *B_SCTRL* process comprises two process equations for each state. The first process models the execution of ASL statements in the state's entry action. In *B_Off_ENTRY* this corresponds to obtaining the light to which the button is attached via *R1*, and generating a *LL2_switchOffSignal* communication. In *LL_On_ENTRY*, defined in Figure 9, we set the `status` attribute which means calling the operation corresponding to the communication along *LL_default_set_status*[2].

The second process defined for a state captures the effects of a signal in that state. e.g., *LL_On_STATE*. Our transformation rules for signal effects are described in Table 3. Signals that cause a transition to another state are removed from one of the object's signal queues via *remove*, after which the successor's entry action is performed. We ensure signals with the *ignored* effect have no consequence and are removed from the signal queue, before returning to the same process; and *cannot happen* signals give rise to a *msg.ih.cannot_happen* event, where *ih* is an instance handle.

4.3 Generating the execution environment

In order to analyse the generated CSP || B specifications we must consider their execution environment. We presented our original version in [16] but the model

[2] In general, we need to distinguish between object a's methods of class A being called by a itself and by other objects. Synchronisation must only occur between the methods in $A\_DOPS(a)$ and their occurrence in other *SCTRL* processes, and not between those found in any of a's *_ENTRY* processes.

$L\_SCTRL(ll) = \ldots$
$\quad LL\_On\_ENTRY = LL\_default\_set\_status!ll!true \rightarrow LL\_On\_STATE$
$\quad LL\_On\_STATE =$
$\qquad remove!ll?\_!LL2\_switchOffSignal \rightarrow LL\_Off\_ENTRY$
$\qquad \square$
$\qquad remove!ll?\_!LL2\_switchOnSignal \rightarrow msg.ll.ignore \rightarrow LL\_On\_STATE$

Fig. 9. Fragment of generated CSP for the Light state chart

Description	CSP Translation
Transition from *state* to successor *succ* on occurrence of signal, *sig*	$remove!ih?\_!sig \rightarrow \langle c\rangle\_\langle succ\rangle\_ENTRY$
Ignored signals, *igs*, are consumed	$\square_{sig \in igs}\ remove!ih?\_!sig \rightarrow msg.ih.ignore \rightarrow \langle c\rangle\_\langle state\rangle\_STATE$
Cannot happen signals, *chs*, trigger cannot happen events	$\square_{sig \in chs}\ remove!ih?\_!sig \rightarrow msg.ih.cannot\_happen \rightarrow \langle c\rangle\_\langle state\rangle\_STATE$

Table 3. Transformation rules for signal effects

presented was not general enough. Consider the following:

$$P1 = generate!i1!i3!s1 \rightarrow generate!i1!i2!s2 \rightarrow STOP$$
$$P2 = remove!i2?s \rightarrow generate!i2!i3!s \rightarrow STOP$$
$$P3 = remove!i3?s1 \rightarrow remove!i3?s2 \rightarrow STOP$$

Assuming no other processes, $P2$ will only generate its signal after it has received $P1$'s $s2$ signal. Hence, using our original definition of the queues, $P3$ will necessarily receive $P1$'s $s1$ signal before $P2$'s $s2$ signal, but this is too deterministic. The rules stated in Mellor & Balcer [2] do not enforce this: there is nothing to prevent $P3$ getting $P2$'s signal before $P1$'s. Our solution was to change the queuing model: for each instance i there is a queue, SQ, to handle self generated signals and also a queue, Q, for each instance j that is different from i, and which describes the behaviour of i with respect to the signals generated from j to i. Furthermore, queues must only associated with active objects. Thus, all the queues can be collectively defined as *SignalQueues* (renaming create/delete omitted) and are initially empty as follows:

$$\big|\big|\big|_{i \in INSTANCE}\ create.i \rightarrow ((SQ(i, \langle\rangle) \underset{\{|remove.i|\}}{\|} (\big|\big|\big|_{j \in (INSTANCE - \{i\})}\ Q(i, j, \langle\rangle))) \triangle (delete.i \rightarrow STOP))$$

4.4 Translating supporting ASL to enable animation

Besides the CSP processes representing the state charts of classes, our tool also generates initialisation segments and test methods as processes, e.g., *InitSegment* and *TestMethod*. This enables us to define an animation scenario in terms of the following:

$$((SYSTEM \underset{\{|externalGenerate|\}}{\|} TestMethod) \parallel InitSegment)$$

where *SYSTEM* represents the controllers of the specification and the signal queues, and the *externalGenerate* channel enables the test method to invoke signals that are externally visible from within the model. For example, a *TestMethod* defined as $externalGenerate.b1.ext0.B1\_buttonOnSignal \rightarrow STOP$ generates a $B1\_buttonOnSignal$ from the external instance $ext0$ to the button instance $b1$. Our tool creates a specified number of animation scenarios for each model.

5 Towards an analysis framework

The contributions of this paper beyond that of the original transformation strategy presented in [16] are the mapping of the creation and deletion of objects, the correction and generalisation in the queuing model, the inclusion of classes that do not have state charts, initialisation segments and test methods. These were described in the last section and are implemented in our tool. In this section we discuss an analysis framework that is work in progress. We identify three kinds of analysis of interest: model consistency checking, deadlock freedom checking, and the verification of the Effects table. In the KC tool validation takes place via simulation, which will only expose bad behaviour if the right animation scenario is provided. Our framework aims to verify that under no circumstances such bad behaviour is possible. We have yet to consider other system properties that could be specified using LTL formulae in ProB and also CSP specifications. For example, verifying that if the button is switched on the corresponding light status is eventually on could be achieved by observing the signal event and then checking the status attribute value.

5.1 Analysis of the Effects table

This first analysis is straightforward; we define the following specification:

$$NoCannotHappens = \Box_{s \in (\Sigma - \{| nomsg |\})} s \rightarrow NoCannotHappens$$

where $\{|\ nomsg\ |\}$ is the set of all messages involving *cannot_happen*. If the following check holds:

$$NoCannotHappens \sqsubseteq_T ((SYSTEM \underset{\{| externalGenerate |\}}{\parallel} ExternalSignals)$$

where *ExternalSignals* represents a recursive choice over all externally generated signals for active objects, then we can be confident that *cannot_happen* communications are not possible in specification. It only makes sense to label *cannot_happen* messages on internal signals. The refinement is trivially false if external signals are labelled *cannot_happen* since they can be invoked in any state.

5.2 Model consistency

Model consistency means checking that a model preserves its multiplicities at certain execution points. Instances can be created independently and a subsequent explicit `link` statement sets up the association between them. Thus, for example, a 1..1 association would not be preserved until after the `link` statement has been executed. However, the validity of the association is only important at the point a navigation occurs. Otherwise, an action may attempt to navigate to an invalid instance.

Associations are dynamic since object and relationship management ASL statements can be used within entry actions, and so model consistency needs to be checked throughout the execution of a system. Our current work is investigating a rely/guarantee style for consistency checking, which spans across state charts and initialisation segments and will be based upon our work on *decompositional verification* in CSP $\parallel$ B [6]. This is where we see a benefit of explicitly modelling association information in the *ASSOC* machine.

The way to proceed is to identify for each state action A_i in each state machine P the predicate that needs to be guaranteed in order to prevent model inconsistency. This predicate is then attached to all the incoming signals of A_i as a *blocking assertion* (which is a rely condition). We then find appropriate associated state actions $Q_1 \ldots Q_n$ which ensure that the assertions are guaranteed, and decorate the relevant `generate` signals with *diverging assertions* (which are guarantee conditions). We would need to demonstrate, using our weakest precondition control loop invariant (CLI) technique, that the process expressions related to each A_i's entry action in P's CSP controller is model consistent with respect to P_MCH and $ASSOC$. That is, $(P\_A_i\_Entry \parallel P\_MCH \parallel ASSOC)$ is divergence-free meaning the guarantees have fulfilled the rely conditions. In general, it is not the case that diverging assertions and their corresponding blocking assertions are on the same signal, and in some cases the rely/guarantee conditions are contained and fulfilled within a single entry action [6].

In our example, there is only one state action that contains behaviour which could result in the model being inconsistent: the `On` state action of the *Button* state chart. Thus, we can identify from the **navigate_R1_from_B** operation the predicate $b \in \mathsf{ran}(R_1)$ as the *blocking assertion* on the *buttonOn* signal. (We would need to be careful how variables were quantified, i.e., $b \in B\_INSTANCES$.) Normally, we would find the corresponding generating signals and attach *diverging assertions*, provided that the signal was generated internally by the model. Decorating signal channels with assertions and their associated proofs ensures that the pair of instances exists when the signal is generated. Subsequently, when we retrieve the light instance in the `On` state it is a valid instance. In our example the signal generation of *buttonOn* is externally controlled, and therefore, the blocking assertion becomes an assumption of the external environment; and anyone using this xUML model would need to discharge it. As we have seen, the *buttonOn* signal can be invoked after the initialisation segment. Therefore, we would need to demonstrate that all operation sequences resulting from the $InitSegment \parallel BUTTON\_MCH \parallel LIGHT\_MCH \parallel ASSOC$ establishes the identified predicate.

Only a single predicate needs to be relied upon in our example. However, this need not be the case in general; it can be a conjunctive predicate identified from propagating all the conditions which need to be relied upon within a state action.

Thus, we may be required to demonstrate that more than one complementary action or external method ensures that such predicates are preserved to ensure model consistency.

5.3 Deadlock Freedom Checking

Deadlock checking means that the model does not deadlock with respect to the processing of signals in active objects. For example, we need to check that:

$$((\mathit{SYSTEM} \underset{\{|\mathit{externalGenerate}|\}}{\|} \mathit{ExternalSignals}) \parallel \mathit{InitSegment})$$

is deadlock-free. It follows a similar pattern to an animation scenario but here the *ExternalSignals* process replaces a particular test method in order to allow always the availability of externally generated signals for active objects. However, this only gives us assurance that the system does not deadlock in the context of a particular object initialisation. We need to provide confidence that the system does not deadlock given an arbitrary valid collection of active objects. This means that we need mechanisms for generating collections of objects that preserve the consistency of multiplicities described in the class diagram, using the same notion of consistency as above. It makes no sense to check for deadlocks when the multiplicities are not preserved at certain execution points. We have developed *multiplicity templates* for all the association types supported in xUML.

Recall that button instances in our example are uniquely related to light instances via the $R1$ association. Therefore, before a button instance can perform a communication along the channel *navigate_R1_from_B*, in order to find its corresponding light instance, it must have been associated with it via a *link* event. Furthermore, button instances cannot be related to other instances unless they have first been unlinked; instances can only be deleted if they are no longer connected to other instances. Figure 10 provides a CSP definition of a process which constrains button instances to link to precisely one light instance (in order to identify the 1 multiplicity at the light end of $R1$ association in Figure 1). A similar template is needed for light instances and together they provide the overall multiplicity constraint.

We then construct a parallel process, $\mathit{CONSTRAINED\_CTRLS}$, which is the combination of the button and light controllers constrained by their multiplicity templates:

$$(\mathit{BUTTONOBJECTS\_R}1 \parallel \mathit{B\_CTRLS}) \parallel (\mathit{LIGHTOBJECTS\_R}1 \parallel \mathit{LL\_CTRLS})$$

The final system, $\mathit{SYSTEM}2$, is the parallel composition of these constrained controllers, together with their signal queues, and the process which enables external signals to be generated, defined as follows (renaming has been omitted):

$$\mathit{CONSTRAINED\_CTRLS} \underset{\{|\mathit{create},\mathit{delete},\mathit{generate},\mathit{remove}|\}}{\|} \mathit{SignalQueues} \underset{\{|\mathit{create},\mathit{delete},\mathit{externalGenerate}|\}}{\|} \mathit{ExternalSignals}$$

Without the inclusion of the *ExternalSignals* process the system would not be able to evolve the behaviour of the button and light state charts.

Finally, we define a specification, SPEC, which is a composition of the *ExternalsSignals* process and processes which represent conditional deadlocks

$$
\begin{array}{l}
\mathit{ButtonMT\_R}1(b) = \\
\quad \text{let } \mathit{ButtonMT\_1} = \mathit{B\_create}!b \rightarrow \mathit{link\_R}1?a!b \rightarrow \mathit{ButtonMT\_2}(\{a\}) \\
\qquad \mathit{ButtonMT\_2}(\varnothing) = \\
\qquad\qquad (\mathit{B\_delete}!b \rightarrow \mathit{ButtonMT\_1}) \;\Box \\
\qquad\qquad \mathit{link\_R}1?a!b \rightarrow \mathit{ButtonMT\_2}(\{a\}) \\
\qquad \mathit{ButtonMT\_2}(S) = \\
\qquad\qquad \mathit{unlink\_R}1?a \in S!b \rightarrow \mathit{ButtonMT\_2}(S - \{a\}) \;\Box \\
\qquad\qquad \mathit{navigate\_R}1\mathit{\_from\_B}!b?a \in S \rightarrow \mathit{ButtonMT\_2}(S) \\
\quad \text{within } \mathit{ButtonMT\_1} \\
\mathit{BUTTONOBJECTS\_R}1 = (\mathop{|||}\, b \in \mathit{Button\_INSTANCES} \bullet \mathit{ButtonMT\_R}1(b))
\end{array}
$$

Fig. 10. Button Multiplicity Template for $R1$

for both buttons and lights. An object cannot deadlock when it is active. The form of such a process for a button i is as follows:

$$
\begin{array}{l}
\mathit{B\_CD}(i) = (\mathit{B\_create}.i \rightarrow \mathit{B\_CDF}(i)) \sqcap \mathit{STOP} \\
\mathit{B\_CDF}(i) = (\mathit{B\_delete}.i \rightarrow \mathit{B\_CD}(i)) \sqcap (\bigsqcap_{b \in B(i)} b \rightarrow \mathit{B\_CDF}(i))
\end{array}
$$

where $B(i)$ is the set of all events that the active object i can engage in. If $\mathit{SPEC} \sqsubseteq_F \mathit{SYSTEM}2$ holds then the specification is shown to be deadlock-free. The theoretical foundations of CSP ‖ B [1] allow us to deduce that if the controllers of a specification are deadlock-free then the whole specification is deadlock-free. Hence, the above deadlock check allows us to conclude that $\mathit{SYSTEM}2 \parallel (\mathit{LIGHT\_MCH} \parallel \mathit{BUTTON\_MCH} \parallel \mathit{ASSOC})$ is deadlock-free. We anticipate that in order to extend this to apply to several relations for each class we will define a separate multiplicity template for each association of a class and the $\mathit{CONSTRAINED\_CTRLS}$ process will be the composition of each controller and all its associated multiplicities. The controllers themselves can already support manipulating several associations. The open research issue is whether will we need to develop decomposition arguments in order to make the model checking tractable.

6 Conclusions

This paper outlined our tool that produces CSP ‖ B specifications and which is integrated into the KC toolset. The specifications are based on transformation rules that are invoked during the traversal of the xUML meta-model using particular xUML classes and any associated state charts. The tool can translate the structure of class diagrams and state charts with limited object manipulation. We have documented assumptions and other forms of restrictions on the xUML input models. More work is needed to identify what datatype definitions cannot be supported and this issue is also identified in [17] as a potential weakness. The subset we use will need to be sufficiently wide ranging so that the translated models can be verified using tool support.

There is a significant body of work relating UML and formal methods, including [7, 8]. The action language in Snook and Butler's UML-B tool is more abstract than ASL, and guards are used within Event-B to provide the control instead of CSP. The emphasis of the UML-B tool is to provide a graphical interface to Event-B rather than analysing the integrity of a UML model. As far as we know, they do not provide support for instrumenting a formal model with animation scenarios based on initialisation segments and test methods. Larsen's work [8] on mapping VDM++ from UML is very pragmatic. The specifier is able to add to the VDM++ produced and these additions are preserved, even if the specification is re-generated. Furthermore, errors from the VDM++ can be mapped back to the UML and this is an important feature in order to provide backwards traceability.

References

1. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Formal Asp. Comput. **17**(4) (2005) 390–422
2. Mellor, S.J., Balcer, M.J.: Executable UML, A Foundation for Model-Driven Architecture. Addison-Wesley (2002)
3. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Epsilon development tools for Eclipse. In: Eclipse Summit. (2006)
4. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
5. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
6. Schneider, S.A., Treharne, H., Evans, N.: Chunks: Component verification in CSP || B. In: IFM. Lecture Notes in Computer Science (2005) 89–108
7. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1) (2006) 92–122
8. Group, T.V.T.: The Rose-VDM++ link. Technical report, CSK Systems (2008)
9. Wilkie, I., King, A., Clarke, M., Weaver, C., Raistrick, C., Francis, P.: UML ASL Reference Guide (ASL language level 2.5). Kennedy Carter Ltd. (2003)
10. Object Management Group: UML 2.0 OCL Specification (2003)
11. Formal Systems Oxford: FDR 2.83 manual (2007)
12. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: FM 2003. LNCS 855–874
13. Schneider, S.: Concurrent and Real-Time Systems: the CSP Approach. Wiley (1999)
14. Roscoe, A.W.: The theory and practice of concurrency. Prentice Hall (1998)
15. Evans, N., Treharne, H., Laleau, R., Frappier, M.: Applying CSP || B to information systems. Software and System Modeling **7**(1) (2008) 85–102
16. Treharne, H., Schneider, S., Grant, N., Evans, N., Ifill, W.: A step towards merging xUML and CSP || B. In: Dagstuhl workshop on Rigorous Methods for Software Construction and Analysis. (to appear)
17. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: MoDELS. Volume 4735 of LNCS. (2007) 436–450