An Optimization Approach for Effective Formalized fUML Model Checking

Islam Abdelhalim, Steve Schneider and Helen Treharne

Department of Computing, University of Surrey

i.abdelhalim, s.schneider, h.treharne @ surrey.ac.uk

Abstract. Automatically formalizing fUML models into CSP is a challenging task. However, checking the generated CSP model using FDR2 is far more challenging. That is because the generated CSP model holds many implementation details inherited from the fUML model, as well as the formalization of the non-trivial fUML inter-object communication mechanism. Using the state space compression techniques available in FDR2 (such as Supercompilation and compression functions) is not enough to provide an effective model checking that avoids the state explosion problem. In this paper we introduce a novel approach that makes use of a restricted CSP model (because it follows certain formalization rules) to optimize the generated model. As an application of our approach, we design a framework that works on two levels; the first one provides optimization advice to the modeller, while the second one automatically applies optimization rules which transform the CSP model to a more optimized one.
Implementing and applying the approach on two large case studies demonstrated the effectiveness of the approach. We also proved that the optimization rules are safe to be applied automatically without eliminating important information from the CSP model.

1 Introduction

Formalizing fUML (Foundational Subset for Executable UML) [1] models to a CSP (Communication Sequential Processes) [2] formal representation, has been previously considered [3, 4]. Having many implementation details in the fUML models and considering the asynchronous inter-object communication mechanism of the fUML standard, always leads to convoluted CSP models. Checking those models using FDR2 [5] is a very challenging task due to the high possibility of the state space explosion problem, or even the length of time taken to perform the model checking.

In this work, we do not introduce a new optimization algorithm that can be applied on the state space level because we do not have internal access to FDR2.

Instead, we propose an optimization approach that works on two levels. Firstly, on the fUML model level, where fUML optimization rules ("fUML-Opti-Rules") are applied to provide the modeller with advice to optimize his fUML model. Secondly, on the CSP model level, optimization rules ("CSP-Opti-Rules") are applied to generate another more optimized CSP model. We propose a framework that applies this approach, beside the formalization tasks. The fUML-Opti-Rules are applied automatically by the Optimization Advisor component which generates some directions (advice) to the modeller that guides him to do the optimization manually, while the Model Optimizer component applies the CSP-Opti-Rules to generate an optimized CSP model

Our optimization is not to be applied on any CSP model (not generic optimization rules), because it is applied on a generated CSP model that follows certain formalization rules (defined in [3]) and built of a subset of the CSP language. The main contribution of this work is that we seize the opportunity of having such a constrained CSP model to develop optimization rules that lead to a very reduced state space. We will proof also that having a specialized optimization rules that are valid for certain property (deadlock freedom) can boost the optimization to new areas that are hard to be reached with the generic ones.

We implemented this framework as a MagicDraw [1] [6] plugin called "Compass" to allow modellers to seamlessly use our framework during the system modelling process. The framework is based on Epsilon [7] to do the MDE (Model Driven Engineering) tasks, such as the model validation and the model-to-model transformation, supported by the fUML and CSP meta-models available in [1] and [8] respectively.

In order to realize and validate our approach, we modelled the GSS (Gas Station System) [9] and the CCS (Cruise Control System) [10] case studies in fUML, and then we used Compass to formalize, optimize and then check the model using FDR2. The GSS fUML model consists of nine objects communicating with each other asynchronously, while the CCS consists of seven objects. The behaviour of each object is modelled as an fUML activity diagram, parts of three of them are included in this paper. The results [2] of applying the optimization advice and rules are outlined across the paper's sections.

The rest of this paper is organized as follows. In Section 2, we give a brief introduction about fUML and CSP. In Section 3, we give an overview about the framework that applies our optimization approach. In Section 4, Section 5 and Section 6 we describe the Optimization Advisor, Model Formalizer and Model Optimizer components respectively of this framework. Finally, we discuss related work and conclude in Sections 7 and 8 respectively.

[1] MagicDraw is an (award-winning) architecture, software and system modeling case tool. It also supports additional plugins to increase its functionalities.

[2] All the model checking in this paper has been done on an Intel Core 2 Duo machine with 2 GB memory.

2 Background

2.1 fUML

As defined by the OMG, fUML [1] is a standard that acts as an intermediary between "surface subsets" of UML models and platform executable languages. The fUML subset has been defined to allow the building of executable models. Code-generators can then be used to automatically generate executable code (e.g., in Java) from the models. Another option is to use model-interpreters that rely on a virtual machine to directly read and run the model (e.g., fUML Reference Implementation [11]).

The fUML standard includes class and activity diagrams to describe a system's structure and behaviour respectively. Some modifications have been applied to the original class and activity diagrams in the UML2 specification [12] to meet the computational completeness of fUML. The modifications have been done by merging/excluding some packages in UML2, as well as adding new constraints, such as:

- Variables are excluded from fUML because the passing of data between actions can be achieved using object flows.
- Opaque actions are excluded from fUML since, being opaque, they cannot be executed.
- Value pins are excluded from fUML because they are redundant due to the use of value specifications to specify values.

The inter-object communication mechanism in fUML
The inter-object communication in fUML is defined by clause 8 in the standard [1]. Such communication is conducted between active objects only. Active objects in fUML communicate asynchronously via signals (kind of classifier). This is achieved by associating an object activation with each object that handles the dispatching of asynchronous communications received by its active object. Figure 1 shows the structure related to the object activation.

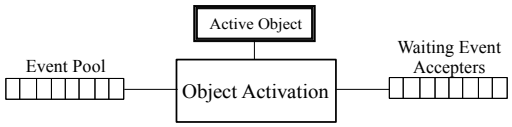


Fig. 1: Object Activation Structure

The object activation maintains two main lists: the first list (called *event pool*) holds the incoming signal instances waiting to be dispatched, and the second list (called *waiting event accepters*) holds the event accepters that have been registered by the executing classifier behaviour. Event accepters are allowable signals with respect to the current state of the active object.

2.2 CSP

CSP [2] is a modelling language that allows the description of systems of interacting processes using a few language primitives. Processes execute and interact by means of performing events drawn from a universal set Σ. Some events are of the form $c.v$, where c represents a channel and v represents a value being passed along that channel. Our fUML formalization and the optimization theorems proofing consider the following subset of the CSP syntax:

$$\begin{aligned} P ::= & \; a \rightarrow P \;\mid\; c?x \rightarrow P(x) \;\mid\; d!v \rightarrow P \;\mid\; P_1 \,\Box\, P_2 \\ & \mid\; P_1 \sqcap P_2 \;\mid\; P_1 \underset{A\ B}{\parallel} P_2 \;\mid\; P \setminus A \\ & \mid\; let\ N_1 = P_1\ ,\ \ldots\ ,\ N_n = P_n\ within\ N_i \end{aligned}$$

The CSP process $a \rightarrow P$ initially allows event a to occur and then behaves subsequently as P. The input process $c?x \rightarrow P(x)$ will accept a value x along channel c and then behaves subsequently as $P(x)$. The output process $c!v \rightarrow P$ will output v along channel c and then behaves as P. Channels can have any number of message fields, combination of input and output values.

The choice $P_1 \,\Box\, P_2$ offers an external choice between processes P_1 and P_2 whereby the choice is made by the environment. Conversely, $P_1 \sqcap P_2$ offers an internal choice between the two processes.

The parallel combination $P_1 \underset{A\ B}{\parallel} P_2$ executes P_1 and P_2 in parallel. P_1 can perform only events in the set A, P_2 can perform only events in the set B, and they must simultaneously engage in events in the intersection of A and B.

The hiding operation $P \setminus A$ describes the case where all participants of all events in the set A are described in P. All these events are removed from the interface of the process, since no other processes are required to engage in them. The $let \ldots within$ statement defines P with local definitions $N_i = P_i$.

3 The approach framework overview

Our optimization approach has been integrated with our previous formalization framework [4] by adding two extra components, and separating the model-to-text task from the Model Optimizer to another component (CSP Script Generator). Figure 2 shows the comprehensive framework that performs the optimization and the formalization tasks, however we will focus in this paper on the optimization components only.

Initially, the modeller uses a case tool (e.g., MagicDraw) to develop the fUML model of the system and then chooses the property that he wants to check

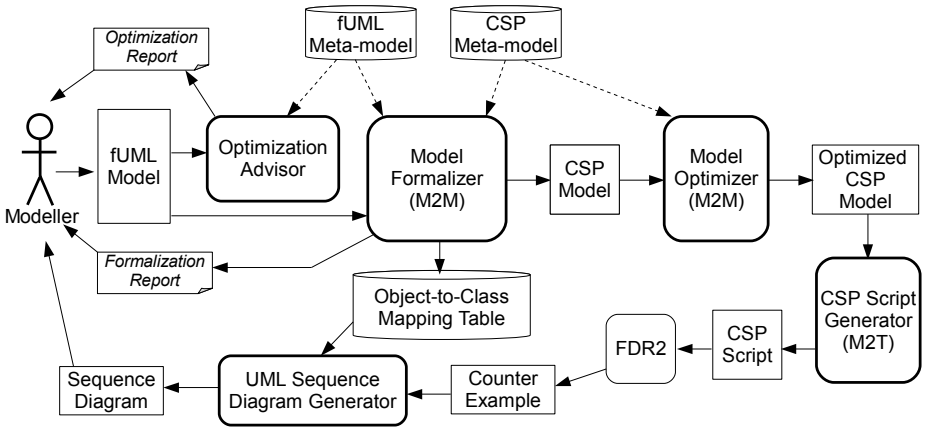


Fig. 2: The Integration of the Optimization Components

(e.g., deadlock freedom). The Optimization Advisor reads the fUML model and searches for specific patterns that if the modeller avoided them, eventually (after the formalization) fewer states will be required to check the CSP model. The advice is reported to the modeller through an Optimization Report, so he can modify the fUML model and start the model formalization.

The Model Formalizer does the model-to-model transformation from fUML to CSP based on their meta-models and a group of formalization rules to generate a CSP model (not script) that represents the fUML model behaviours. It also generates the Object-to-Class mapping table which will be used for traceability by another component. Based on the chosen property, the Model Optimizer starts its function by reading the CSP model and applying a group of optimization rules. Those rules transforms the initial CSP model to an optimized one that still contains the required information to check the chosen property.

The CSP Script Generator component performs the model-to-text task that generates a CSP script from the input optimized CSP model based on the same CSP meta-model. At this point FDR2 can be launched to do the model checking and if the checked property (e.g., deadlock) is not met, FDR2 generates a counter-example. The UML Sequence Diagram Generator reads the counter-example and uses the Object-to-Class mapping table to generate a UML sequence diagram that represents the counter-example in a modeller friendly format.

4 The Optimization Advisor

The fUML model may contain some patterns that are correct from the modeller point of view and the system specification. However, when model checking the CSP representation of this fUML model, a state space explosion problem may

happen. The focus here is on the patterns that cannot be removed automatically because certain decisions will be required from the modeller to avoid them.

The Optimization Advisor component scans the fUML model with regards to those patterns, and if found, it reports advice to the modeller to avoid them. We use EVL (Epsilon Validation Language) [7] together with the fUML meta-model to perform this task. The EVL script consists of a group of constraints, each examines the existence of a certain pattern within a specific context. The following sub-sections describe two fUML-Opti-Rules and their optimization effect.

4.1 fUML-Opti-Rule(1): Detecting self-sending signals

When an object sends a signal to itself, we call this "self-sending". Although this sounds benign, we found by experiment that self-sending signals cause an extra overhead on the object's *event pool* buffer. Generally, a self-sending signal can be replaced by a direct control flow edge that joins the points of sending and accepting this signal. This replacement is safe provided that there is no actions between the two points (as they will be bypassed).

Another case is when the *AcceptEventAction* accepts the self-sending signal beside another signal, such as the one highlighted in Figure 3 for the Pump object activity, because a direct control flow between the *FuelLevelLow* signal sending and acceptance (shown as a dashed line) does not preserve the behavioural semantics of the original activity unless the self-sending signal has the higher priority in the *event pool* (i.e., the first one to be dispatched from the object's *event pool*). For that reason, we do not apply this fUML-Opti-Rule automatically. The Optimization Advisor just highlights the self-sending signals through the Optimization Report, leaving the choice to the modeller to do the removal based on his understanding of the fUML model.

On the other hand, it is obvious that removing the *CustomerFinished* signal sending and acceptance and replacing them with a direct control flow (shown as a dashed line at bottom of the diagram) will not affect the overall behaviour of the object.

We have developed an EVL constraint to check this pattern and report the advice to the modeller. The following EVL constraint applies fUML-Opti-Rule(1) on any *SendSignalAction*. The *if* condition checks if the target object of this send action is the sender object. The *message* field defines what the modeller will see in the Optimization Report in case that this constraint was not met.

```
context ActivityDiagram!SendSignalAction {
  constraint fUML-Opti-Rule-1{
  check {
   if(self.target.incoming.source.type.name = self.owner.name){return false;}
   else{ return true; } }
  message : "The signal action '" + self.name + "' sends the signal to its
            object which can be replaced by a direct control flow." }}
```

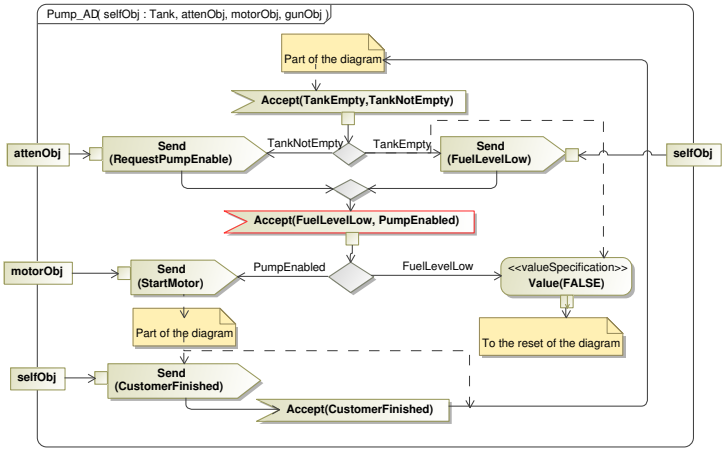


Fig. 3: Part of the Pump object activity

The Optimization Advisor managed to detect 4 self-sending signals in the GSS fUML model, and by safely replacing them with direct control flows the state space was reduced from 10.2M states to 4.7M states (where M stands for million) when checking the CSP formal representation of that fUML model.

4.2 fUML-Opti-Rule(2): Detecting unacknowledged signals

An "unacknowledged" signal, is one that has been sent from an object to another object, and then it (source object) continues sending further signals without waiting for an acknowledgment signal. The problem arises when this pattern is repeated several times, because the system will be flooded with the unacknowledged signals and thus the objects' *event pool* buffers will overflow.

As an example in the GSS, the Meter object activity was doing nothing but sending the *FuelUnitDelivered* signal to the Delivery object causing its *event pool* to overflow very fast. If we acknowledged this signal by adding an accept event action for the *FuelUnitDeliveredACK* signal after the sending action, the Meter object will be forced to wait until the Delivery object sends the acknowledgment.

The Optimization Advisor uses the fUML-Opti-Rule(2) (represented in EVL) to scan the fUML model searching for the unacknowledged signals and report their existence through the Optimization Report. It is the modeller responsibility to acknowledge the signals in the fUML model.

fUML-Opti-Rule(2) helped in detecting three unacknowledged signals in the GSS fUML model. Acknowledging those signals reduced the state space of the corresponding CSP model from 4.7M to 3.0M states when checking its deadlock freedom.

5 The Model Formalizer

The Model Formalizer's main function is to automatically transform the fUML model to a CSP formal representation that captures its behaviour. The transformation is done using ETL (Epsilon Transformation Language) which requires the source fUML meta-model and the target CSP meta-model available in [1] and [8] respectively. In our previous work [3] we presented a group of formalization rules which maps between the fUML activity diagram elements and its corresponding CSP. In [4] we described how we used a model-to-model transformation technique to apply those formalization rules automatically using ETL. In this paper we are using the same formalization approach with additional formalization rules to be able to formalize the GSS fUML model.

Our focus in this paper is on the optimization rather than the formalization, so we will not discuss the formalization rules and limit our discussion to the final result (CSP model) when applying the formalization rules on an fUML activity diagram. As an example for this application, consider the Tank object activity diagram in Figure 4. The diagram shows part of the activity which initially waits for the *FuelUsed* or the *RequestTankStatus* signals. If the Tank object accepted the *FuelUsed* signal, it subtracts the received amount from the current *tankLevel* by executing the *ReduceLevel* CallBehaviour action, then it checks the current level to set the *tankEmptyFlag* to *TRUE* if the level is below a certain threshold. If the Tank object accepted the *RequestTankStatus* signal (as a query from the Pump object), the *CheckLevel* CallBehaviour action is executed so the objects sends a *TankEmpty* signal in case where the action returned *TRUE*, otherwise it sends *TankNotEmpty* signal.

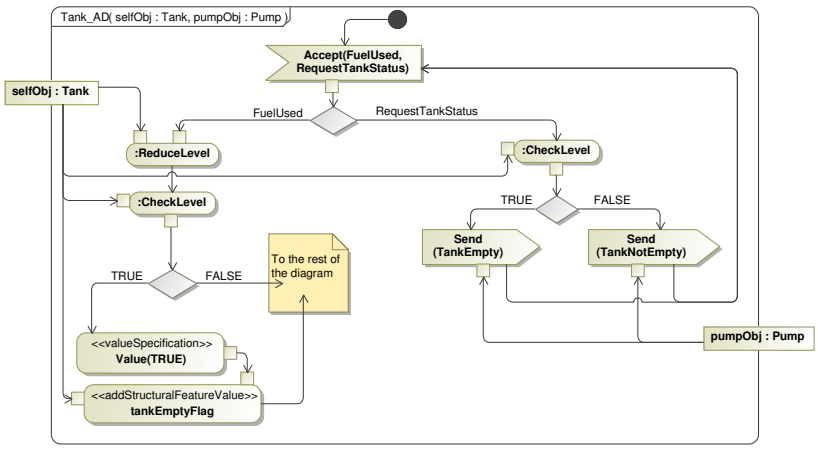


Fig. 4: Part of the Tank object activity

The Model Formalizer generates a CSP model by applying the ETL formalization rules on the Tank fUML activity diagram. This CSP model can be represented as the following CSP localized process that captures the Tank activity behaviour:

$$
\begin{aligned}
&Tank\_AD\_Proc(selfObj, pumpObj) = \\
&let \\
&\quad AC1 = registerSignals!selfObj!rp1 \rightarrow AC3 \\
&\quad AC3 = accept!selfObj!FuelUsed \rightarrow AC2 \\
&\qquad\quad \Box \\
&\qquad\quad accept!selfObj!RequestTankStatus \rightarrow AC4) \\
&\quad AC2 = AC5 \\
&\quad AC5 = AC6 \\
&\quad AC6 = TRUE!selfObj \rightarrow AC7 \\
&\qquad\quad \sqcap \\
&\qquad\quad FALSE!selfObj \rightarrow \cdots \\
&\quad AC7 = valueSpecification!selfObj?var : TRUE \rightarrow AC8(var) \\
&\quad AC8(var) = addStructuralFeatureValue!selfObj!tankEmptyFlag!var \rightarrow ... \\
&\quad AC4 = AC9 \\
&\quad AC9 = TRUE!selfObj \rightarrow AC10 \\
&\qquad\quad \sqcap \\
&\qquad\quad FALSE!selfObj \rightarrow AC11 \\
&\quad AC10 = send!selfObj!pumpObj!TankEmpty \rightarrow AC1 \\
&\quad AC11 = send!selfObj!pumpObj!TankNotEmpty \rightarrow AC1 \\
&within\ AC1
\end{aligned}
$$

The formalization of the fUML model includes also a formalization of the fUML inter-object communication mechanism (described in Section 2.1 and the formalization in [3]) that manages the signals sending, acceptance and dispatching through the *event pool* and the *waiting event accepters* list. The explanation of the localized CSP process below considers the formalization of this mechanism.

Initially, the *registerSignals* event adds the two signals (*FuelUsed* and *RequestTankStatus*) to the *waiting event accepters* list of the Tank object using the registration point *rp1*. In *AC3*, the *accept* event is not enabled until one of the two signals arrives to the object's *event pool*. The internal behaviour of the *ReduceLevel* and *CheckLevel* CallBehaviours will not affect the overall behaviour of the object, for that reason the Model Formalizer abstracts them into the processes *AC2*, *AC5* and *AC4* (whose just enabling the next sub-process), and converts the decision based on their outputs to an internal choices (in *AC6* and *AC9*) so the model checker explores all the possible outputs. *AC7* and *AC8* are direct formalization for the *ValueSpecificationAction* and the *AddStructuralFeatureAction* respectively. Finally, the *send* event synchronizes with the other objects to insert signals in their *event pools* (e.g., *TankEmpty* in the Pump object's *event pool*).

6 The Model Optimizer

After the Model Formalizer finishes its job, the Model Optimizer starts the automatic optimization of the generated CSP model. The Model Optimizer performs

a further transformation to that CSP model so that the model checker (FDR2) requires less states and time to check it. This transformation is done by applying group of optimization transformation rules (CSP-Opti-Rules). The CSP-Opti-Rules are defined in ETL also to perform such model-to-model transformation task based on the same CSP meta-model. Because the CSP-Opti-Rules are applied automatically, we support each one with a mathematical proof (theorem) to make sure that it does not eliminate important information that is required for a specific property checking (e.g., deadlock).

As mentioned in the introduction, our CSP-Opti-Rules are not generic (i.e., they cannot be applied on all CSP models). We make use of the opportunity of having a constrained CSP model that has been generated from specific rules that consider only a subset of the CSP language. Also, our CSP-Opti-Rules are constrained with the checked property. For example, CSP-Opti-Rule(2) can be applied if and only if the modeller wants to check the deadlock freedom.

The following sub-sections show three CSP-Opti-Rules and their effect on the state space. The technical report [13] includes the theorems proofs of all the rules.

6.1 CSP-Opti-Rule(1): Removing passive processes

We differentiate between two types of objects. First, core objects, which include the main behaviour of the system and interact with other objects in both direction (sending and accepting signals). Second, terminal objects, which represents external entities; however they interact with the system. For example, the GSS includes an object for the Attendant just to simulate his interaction with the Pump; however, it is a terminal object because it will not be part of the system implementation. To check the model against deadlock freedom, the modeller should include all kinds of objects (core and terminal) in the fUML model to be able to explore all the system behaviours.

Passive objects are special kind of the terminal objects as they interact in one direction (accepting signals only). The Motor object is one of the obvious example of the passive objects in the GSS as it does nothing but accepting signals (*StartMotor* and *StopMotor* signals) from the Pump.

In the CSP domain, a passive process is defined as the process that represents the passive object behaviour, which is always willing to interact (never refuses any interaction). On the implementation level, CSP-Opti-Rule(1) is represented as an ETL rule that scans the CSP model for any passive process, and if found, removes it from the CSP model. In other words, it removes the passive process from the parallel combination between the system's process which forms the SYSTEM big process. CSP-Opti-Rule(1) rules out a process to be passive if it contains the *send* event, which moves out the process from the passive condition (accepts signals only).

To formally verify that the removal of passive processes will not affect the deadlock checking of the system, we proved the following theorem for the passive process P_2:

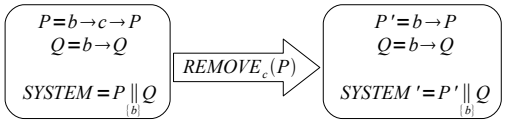
Theorem(1). *If P_2 is non-divergent and $(s, X_{P2}) \in \mathcal{F}(P_2) \Rightarrow X_{P2} = \{\}$ (passive process), then for any process P_1: P_1 is deadlock free $\Leftrightarrow$ $P_1 \parallel P_2$ is deadlock free.*

The assumption that P_2 is non-divergent is guaranteed by the definition of the formalization rules. Refer to [13] for the detailed proof of this theorem.

In the GSS fUML model, there is one passive object (the Motor object) and thus one passive process. When checking deadlock freedom, without applying CSP-Opti-Rule(1) on the corresponding CSP model, FDR2 was unable to complete the check without crashing. After applying CSP-Opti-Rule(1), the Model Optimizer removed the Motor process the CSP model and FDR2 succeeded to report the deadlock freedom after exploring 12.3M states (the whole model).

6.2 CSP-Opti-Rule(2): Removing abandoned events

The global target of this rule is to search the CSP model for any event that can be removed from the model without affecting the deadlock checking results. We identify one kind of events that meets this criteria which we call "abandoned events". Abandoned events are those which do not synchronize with any other events in another processes in the system. The removal is done by skipping to the next event/process as shown in the example below:

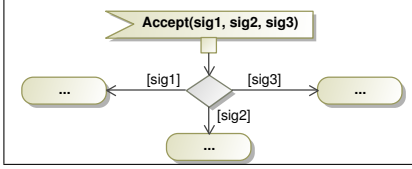


Theorem(2) is the formal representation of CSP-Opti-Rule(2). The theorem shows the constraints that make c an abandoned event, which is not to be member in any other process alphabets (i.e. no process will synchronize on c). It also shows that the removing of c from P_1 ($REMOVE_c(P_1)$) will not affect the deadlock checking result.

Theorem(2). *If $c \in alpha(P_1)$, $c \notin alpha(P_2)$, $P_1 \setminus c$ is non-divergent and $REMOVE_c(P_1)$ is defined then:*
$REMOVE_c(P_1) \parallel P_2$ is deadlock free $\Leftrightarrow$ $P_1 \parallel P_2$ is deadlock free.

The proof of Theorem(2) (available in [13]) is valid only if P_1 is of the form: $P ::= c \rightarrow P \mid c?x!y \rightarrow P(x) \mid P \,\Box\, Q \mid P \sqcap Q$.

Additionally , in the case of $P \,\Box\, Q$, the condition $c \notin initials(P) \wedge c \notin initials(Q)$ should be met, which is guaranteed because we do not use external choices except in the formalization rule below (Rule(4) in [3]):

fUML Element	CSP Representation
Rule(4): Accept Event Action (*) 	$AC1 = registerSignals!bIH!rp2 \rightarrow ($ $accept!bIH!sig1 \rightarrow ...$ $\square$ $accept!bIH!sig2 \rightarrow ...$ $\square$ $accept!bIH!sig3 \rightarrow ...)$

In other words, it is safe to apply CSP-Opti-Rule(2) on any generated CSP model from the Model Formalizer and by the correct selection of the abandoned event (e.g, not to be the *accept* event), provided that the deadlock freedom is the checked property.

We have implemented CSP-Opti-Rule(2) in ETL and applied it on the *valueSpecification* and *addStructuralFeatureValue* events, because they are not synchronizing with any other events and they never happen after an external choice.

The elimination of the abandoned events reduces the state space size, and thus allows faster FDR2 checks. When CSP-Opti-Rule(2) was applied on the GSS corresponding CSP model, 8 abandoned events were removed to reduce the state space from 12.3M to 10.2M states. Using the standard CSP hiding operator instead of $REMOVE_c(P)$ did not provide such reduction in the state space because the hiding does not remove the event, it just renames it to τ.

6.3 CSP-Opti-Rule(3): Toggling internal choices

Unlike the first two CSP-Opti-Rules, this one needs human interaction to be performed. It also does not lead to an optimized version of the original model. Rather, it splits the original model into sub-models that are easier to be checked separately using FDR2. This kind of CSP-Opti-Rules are very useful when analyzing big models, as it will allow the modeller to focus on certain parts of the model at a time. This is a bounded approach to find and solve the model's problems. Another benefit is that when the model is too big to be analyzed by FDR2, the CSP-Opti-Rule can be used to analyze the system on different stages, each stage is an analysis of one of the sub-models.

CSP-Opti-Rule(3) can be summarized as follow: when checking deadlock, if all the sub-models are deadlock free, then the original model is deadlock free as well. The splitting up of the original model is done based, generally, on reducing the behavioural paths in the model's processes. In particular, CSP-Opti-Rule(3) replaces the internal choice with one direct connection to one of its choices. And in case of more than two choices, it disables one of them. The selection of the enabled choice(s) comes from the modeller input through a Graphical User Interface (GUI).

To apply this rule, the Optimization Advisor scans the fUML model for the decision nodes that will be translated to internal choices in the CSP model and build a table with those nodes/choices. After building the table, the modeller can use the GUI to toggle the choice branches and start the model checking. The selected choices will be passed to the Model Optimizer that applies CSP-Opti-Rule(3) based on the modeller selection. After the model checking, the modeller can repeat the process with different choice(s).

Illustrated below Theorem(3), which proofs that this accumulative deadlock checking (on several stages) is equivalent to the original model deadlock checking (as a whole). For example, if the model checking of sub-model-A (which has *decision_1* ON and *decision_2* OFF) is deadlock free, and sub-model-B (which has *decision_1* OFF and *decision_2* ON) is deadlock free, then the whole model is deadlock free.

Theorem(3). *If* $SYS = P_1 \sqcap P_2$, $SYS' = P_1$ *(after splitting* SYS *and choosing the first branch), and* $SYS'' = P_2$ *(after splitting* SYS *and choosing the second branch), then:*
$SYS\ Deadlock\,Free \Leftrightarrow SYS'\ Deadlock\,Free\ \vee\ SYS''\ Deadlock\,Free$

Applying CSP-Opti-Rule(3) on the GSS CSP model by disabling the tank emptiness choice (i.e., assuming that the tank can never be empty), FDR2 managed to check the model for deadlock in 11 minutes after exploring 1.8M states instead of 18.4 minutes and 3.0M states when the two choices are available (i.e., the tank can be empty or not).

7 Related work

There is a significant body of work researched in avoiding the model checking state explosion problem. Some authors such as Planas *et al.* [14] avoided the problem completely by depending on the model static analysis. The others who preferred the model checking, can be categorized as follow:

The first category includes the work that focuses on optimizing the corresponding LTS (Label Transition System) of the formal model. FDR2 (and many other model checkers), works by calculating the LTS semantics of the CSP processes and then perform the model checking on the LTS level [15]. The bisimulation minimization in FDR2 [16] is an optimization technique that lies in this category. The supercompilation [17] is another example where FDR2 calculates a set of rules for turning a combination of LTS's into a single LTS, without explicitly constructing it. Also, Roscoe in [15] showed how to use FDR2 compression functions such as: *sbisim*, *normal* and *diamond* to compress the LTS of the CSP model. ProB [18] is another model checker that uses another optimization techniques on the LTS level such as permutation flooding [19] and hash value [20] symmetric reduction.

The second category includes those who concentrate on the formal model optimization before translating it to the LTS representation. Decomposing the formal

model into constituent parts to have an effective model checking is one of the techniques that has been used in this category. One such example is Wang *et al.* [21] who proposed using Extended Hierarchical Automata (EHA) for UML state diagrams formalization, and then slice the EHA model based on a slicing criterion extracted from the checked property. Another example is Schneider *et al.* [22] who proposed decomposing the CSP||B model into finer grained components called *chunks* to allow checking divergence freedom in large systems using FDR2. Apart from the decomposition, some data abstraction techniques can also be used to optimize the formal model. For example, Jesus *et al.* [23] abstract any infinite domain in the system to allow checking the CSP models using FDR2.

Our work lies in the second category, especially when considering the Model Optimizer component which applies the CSP-Opti-Rules directly. However, we could not find in the literature an approach that provides optimization advice on the semi-formal model level before the formalization. Also implementing a comprehensive framework to apply the formalization and optimization based on an MDA technique is a distinguished point for our work.

8 Conclusion

We have described in this paper a framework for optimizing the formal representation (CSP) of the fUML models. The framework does the formalization and the optimization tasks using different components. We described two of those components that applies the optimization rules. The first one is the Optimization Advisor which uses EVL to provide the modeller with some advice to avoid some undesirable patterns in his fUML model. The second component is the Model Optimizer which uses ETL to generate an optimized CSP model. The following table summarizes the results of applying our approach on the GSS and the CCS case studies using Compass. The "States" field shows the explored number of states by FDR2 until reporting the deadlock freedom of the model.

CSP-Opti-Rule(1)	CSP-Opti-Rule(2)	fUML-Opti-Rule(1)	fUML-Opti-Rule(2)	CSP-Opti-Rule(3)	GSS		CCS	
					States	Time	States	Time
✓					12.3 M	1.38 H	38.2 M	2.7 H
✓	✓				10.2 M	1.15 H	36.0 M	2.5 H
✓	✓	✓			4.7 M	29.7 Min	25.1 M	1.7 H
✓	✓	✓	✓		3.0 M	18.4 Min	7.1 M	28.5 Min
✓	✓	✓	✓	✓	1.8 M	11.0 Min	5.23 M	20.6 Min

It is obvious that applying the CSP-Opti-Rules and the advice from the fUML-Opti-Rules led to a substantial reduction in the state space and the model checking time. The order of the applied rules is just according to our test scheme. Rules

are independent (i.e., no rule depends on another one) and the more rules the modeller applies the more reduction in the state space he will get.

References

1. OMG: Semantics of a foundational subset for executable UML models (fUML) - Version 1.0 (February 2011)
2. Schneider, S.: Concurrent and Real-Time Systems: the CSP Approach. Wiley (1999)
3. Abdelhalim, I., Sharp, J., Schneider, S.A., Treharne, H.: Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP. In: ICFEM 2010, Shanghai, China, Proceedings. Volume 6447 of LNCS., Springer (2010) 371–387
4. Abdelhalim, I., Schneider, S.A., Treharne, H.: Towards a Practical Approach to Check UML/fUML Models Consistency Using CSP. In: ICFEM 2011, Durham, UK, Proceedings. Volume 6991 of LNCS., Springer (2011) 33–48
5. Formal Systems Oxford: FDR 2.91 manual (2010)
6. MagicDraw CASE tool. http://www.magicdraw.com/
7. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book
8. Treharne, H., Turner, E., Paige, R.F., Kolovos, D.S.: Automatic Generation of Integrated Formal Models Corresponding to UML System Models. In: TOOLS (47). (2009) 357–367
9. Chris Raistrick, Paul Francis, J.W.C.C.I.W.: Model Driven Architecture with Executable UML. Cambridge University Press (2004)
10. Cameo Simulation Toolkit. https://www.magicdraw.com/simulation
11. ModelDriven.Org: fUML Reference Implementation. http://portal.modeldriven.org (last viewed 4th of October 2011)
12. OMG: Unified modeling language (UML) superstructure (version 2.3) (2010)
13. Abdelhalim, I., Schneider, S.A.: Optimization Rules Mathematical Proofs. Technical Report, University of Surrey (October 2011)
14. Planas, E., Cabot, J., Gómez, C.: Verifying action semantics specifications in UML behavioral models. In: CAiSE '09: Proceedings of the 21st International Conference on Advanced Information Systems Engineering, Berlin, Heidelberg, Springer-Verlag (2009) 125–140
15. Roscoe, A.W.: Understanding Concurrent Systems. Springer (2010)
16. Roscoe, A.W.: The Theorey and Practice of Concurrency. Prentice Hall (1998)
17. Goldsmith, M.: Operational Semantics for Fun and Profit. (2005) 265–274
18. The ProB Animator and Model Checker. http://www.stups.uni-duesseldorf.de/ProB
19. Leuschel, M., Butler, M., Spermann, C., Turner, E.: Symmetry reduction for B by permutation flooding. In Julliand, J., Kouchnarenko, O., eds.: 7th International B Conference. Volume LNCS 4., Springer (January 2007) Copyright Springer.
20. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: The International Symmetry Conference. (2007)
21. Wang, J., Dong, W., Qi, Z.C.: Slicing hierarchical automata for model checking UML statecharts. In: ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods, London, UK, Springer-Verlag (2002) 435–446
22. Schneider, S., Treharne, H., Evans, N.: Chunks: Component Verification in CSP||B. In: IFM'2005. (2005) 89–108
23. Jesus, J., Mota, A., Sampaio, A., Grijo, L.: Architectural Verification of Control Systems Using CSP. In: ICFEM. (2011) 323–339