Combining Static and Dynamic Techniques for Refactoring Industrial FSMs in C++

Mathijs Schuts[1], Jozef Hooman[2] and Marco Alonso[1]

[1]Philips, Best, The Netherlands
[2]TNO-ESI, Eindhoven, The Netherlands

Abstract

Software maintenance requires a significant amount of time and effort. One of the tasks involved in this process is to replace outdated frameworks with newer, state-of-the-art alternatives. For such a replacement, static and dynamic techniques can be utilized. Using a static technique, such as metaprogramming, the source code can be inspected in a whitebox manner to gain insight in its functionality. By means of a dynamic technique, such as Active Model Learning (AML), the behavior of the code can be extracted in a blackbox manner. Since both techniques have their advantages and disadvantages, we propose a novel combination of static and dynamic analysis techniques that complements each other. Model checking is used to check the equivalence of the models obtained by the different techniques. The approach has been applied at Philips to upgrade a legacy framework written in C++ to describe Finite State Machines (FSMs). This framework has been replaced by a more modern tool called Dezyne. By means of our new approach, we were able to semi-automatically replace 14 FSMs with high confidence in the preserved behavior.

Keywords

Refactoring, Metaprogramming, Domain Specific Language, Finite State Machine, C++, Abstract Syntax Tree, Active Model Learning, Model Checking, Equivalence Checking, Model Based Development, Industry Case

1. Introduction

High-tech systems have numerous components that serve various purposes, and they are often old or custom-made. Maintaining these legacy components can be a significant burden on developers [1]. Modern software engineering practices offer alternatives to rudimentary coding methods for complex systems like supervisory control software. These alternative approaches include Domain Specific Languages (DSLs) that allow intuitive expression of state machines, improving the overall quality and maintainability of high-tech systems [2].

Working with both a legacy and a new framework for components can make maintaining a large code base more difficult. Replacing an old framework with a newer one enhances maintenance efficiency. The majority of effort is in refactoring existing components to use the new framework instead of the obsolete one, but ensuring behavior preservation after changes requires additional effort.

The challenge lies in replacing only outdated framework calls without altering other components' code. Regression test suites usually do not provide sufficient confidence that behavior has been preserved because of limited code coverage. Hence, the challenge is to ensure that changes have been implemented correctly and preserve the original behavior.

Static techniques like metaprogramming can analyze software internals but cannot extract meaning; dynamic techniques observe semantics but lose internal information. The Rascal metaprogramming language [3] is an example of a static technique that considers source code as data, allowing it to identify, for instance, the names of internal functions. It achieves this by parsing the source code into an Abstract Syntax Tree (AST), which can be used to construct a model like a Finite State Machine (FSM) representing the behavior of a piece of source code. Static techniques, however, cannot extract the execution semantics from the source code [4]. Hence, the use of metaprogramming for transformations faces challenges in determining the precise FSM semantics [5].

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium
mathijs.schuts@tno.nl (M. Schuts)

Dynamic techniques can execute the source code and generate a behavioral model that can be used to check if a new implementation has the same behavior as the legacy implementation. Active Model Learning (AML) [6] is an example of a dynamic technique that can be applied to construct behavioral models from existing software. Figure 1 shows how a learner application interacts with the software for which a model has to be made, the so-called System Under Learning (SUL).

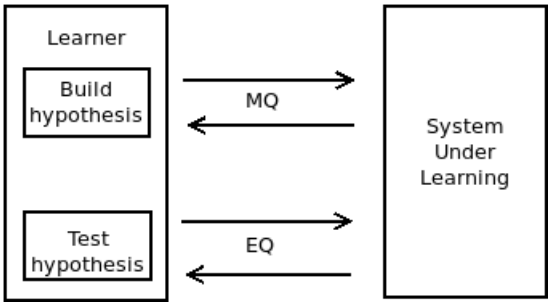

Figure 1: Active Model Learning

The learner is provided with the set of possible input events that it can send to the SUL. When learning starts, the learner calls sequences of input events called Membership Queries (MQ) to the SUL. For every sequence of input events, the SUL replies with a sequence of output events. Before a new input events sequence is called, the learner resets the SUL to its initial state. After a number of MQs, the learner constructs a hypothesis which is a state machine model of the SUL behavior. This hypothesis is tested for conformance with the SUL by means of Equivalence Queries (EQ). EQs can either confirm the hypothesis –in this case the hypothesis is the final model– or otherwise it will return a counterexample. The learner uses the counterexample to continue learning and create a new hypothesis. The resulting model is a deterministic finite state machine [7]. There are different learning algorithms available such as L* [7], TTT [8] and L# [9] for MQs and, i.e., Wp [10] and I-ADS [11] for EQs.

In Table 1, we compare static and dynamic techniques. It indicates that a combination of both techniques offers additional benefits over the use of each of them individually.

Technique	Advantage	Disadvantage
Static	Internal structure of software can be inspected.	Execution behavior might be different.
Dynamic	Behavior during execution.	Cannot inspect the internals of software.

Table 1
Comparing static and dynamic techniques

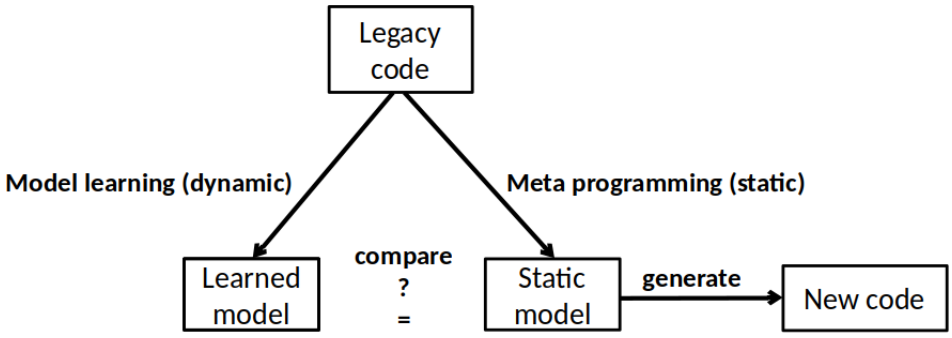

Figure 2: Overview of the combined approach

Figure 2 shows how we integrate the two approaches to acquire models of legacy code. We use the mCRL2 model checker to check the equivalence of the two models [12]. Note that when two finite state machines do not contain any unobservable internal transitions, trace equivalence and branch bisimulation equivalence coincide, and there is no difference between weak and strong equivalences [13].

Our approach is suitable for applications that meet the restrictions of AML which can be applied on data-independent control components, i.e., components where control decisions do not depend on input data. For a successful application of metaprogramming, the legacy code should contain a pattern that can be captured in a metaprogram. To justify the effort in creating such a metaprogram and have a good return of investment, there should be sufficient instances of the pattern.

Industrial application

Our refactoring approach has been applied to an industrial case that has the characteristics described above. It addresses a part of the code of an interventional X-ray system at Philips Image Guided Therapy (IGT). A legacy CppFSM framework is replaced with a new model-based FSM framework called Dezyne [14]. Observe that static and dynamic techniques cover different aspects when refactoring state machines:

- With metaprogramming, we can preserve state names, but ensuring a semantics-preserving behavior is impossible.
- With AML, we can extract the execution semantics, but we lose the state names.

We applied our approach, which combines both techniques, to replace more than a dozen CppFSMs. The combination and an equivalence check increase the confidence that the behavior is preserved after refactoring.

To our knowledge, this paper presents the first instance of combining static and dynamic techniques to refactor industrial C++ code

Structure of the paper

The paper is organized as follows. Background on the used technologies is described in Section 2. In Section 3, we introduce the industrial application. Our approach is highlighted in Section 4. In Section 5, the results of applying our approach to the industrial case are presented. Concluding remarks can be found in Section 6.

2. Background

Related to our work is the book of Fowler on domain-specific languages [15]. Fowler defines refactoring as changing the internals of source code without changing its usage while preserving its behavior. The manual creation of test cases is advocated before the start of the refactoring when the current test cases provide insufficient code coverage. After an improvement of the tests to get better code coverage, the source code can be manually refactored.

As an alternative to manually refactoring code, metaprogramming is a static technique used to analyze and transform source code. A metaprogram accesses source code as whitebox; all internals of the source code can be inspected. Such a static technique cannot acquire the behavior of the source when it is executed [4]. Ivers et al. [16] have also identified that preserving the semantics is a challenge. Especially when dealing with, for instance, function pointers it is difficult to extract the operational semantics [17].

Examples of metaprogramming languages are Rascal [3], Algebraic Specification Formalism + Syntax Definition Formalism (ASF+SDF) [18], Stratego [19] and Turing eXtender Language (TXL) [20]. The following metaprogramming languages are specialized for C++: CodeBoost [21], Design Maintenance System (DMS) [22] and Proteus [23].

These metaprogramming languages have been applied in industry. At Google, the ClangMR [24] has been used to refactor large C++ codebases. They refactored callers of deprecated APIs. The tool is based on the Clang compiler in combination with the MapReduce parallel processor. Mooij et al. [25] refactored C++ code using small iterative steps. After the small code refactoring steps, a model was extracted from the source code before generating a new implementation. Schuts et al. [26] used Rascal

to refactor more than 150 test suites written in C++. They changed API calls to a legacy framework into a new test framework.

The Rascal's library C/C++ Language Analysis In Rascal (ClaiR) can be used to analyze C/C++ code [27]. The conversions written in this paper were all implemented in Rascal. For some we also used ClaiR.

Active Model Learning (AML) has been applied in several non-trivial cases. For instance, to learn models of network protocol implementations, such as SSH, SIP, TCP, TLS, and models of smart cards for banking and bio-metric passports [28]. Model learning is also used in industry, e.g., at Canon to learn a controller of a high-end printing copier of 410 states and 77 stimuli was learned [29].

AML has also been applied in the context of manual refactoring. In [30], we describe a case at Philips where we learned a legacy implementation and a manually refactored implementation. Next we applied an equivalence checker to test if both implementations were equivalent. With this approach, we found issues in both implementations. After solving the issues, the models were equivalent. We also applied model learning and equivalence checking to test a model-to-model transformation from IBM Rhapsody to Dezyne [31].

3. Industrial Application

Because of confidentiality we cannot share models from the interventional X-ray system and instead we explain the techniques using a vending machine example which is introduced in Section 3.1. Section 3.2 presents the CppFSM framework which is the legacy framework we want to remove from the code base. The target Dezyne framework of our transformation is described in Section 3.3.

3.1. Vending Machine Case

Table 2 provides the state machine of the vending machine. The initial state is *idle*. From this state, a two Euro coin can be inserted. When this happens, the vending machine displays select beverage and transitions to the *paid* state. In the *paid* state there are two transitions: one for choosing coffee and yet another self transition for inserting a two Euro coin. On the other hand, coffee can be made black, or with sugar, milk or both. In all states –except from the *idle* state– a cancel event shall return the two Euro coin and go back to the *idle* state. For the readability of Table 2, we omitted the cancel event transitions.

Current state	Input event	Output event	Next state
idle	insertTwoEuroCoin	displaySelectBeverage	paid
paid	coffee	displayCoffeeSelected	coffee
coffee	confirmSelection	MakeCoffee	idle
paid	insertTwoEuroCoin	returnCoin	paid
coffee	selectSugar	displaySugarSelected	coffeeWithSugar
coffee	selectMilk	displayMilkSelected	coffeeWithMilk
coffeeWithSugar	confirmSelection	MakeCoffee	idle
coffeeWithSugar	selectMilk	displayMilkSelected	coffeeWithMilkAndSugar
coffeeWithMilk	confirmSelection	MakeCoffee	idle
coffeeWithMilk	selectSugar	displaySugarSelected	coffeeWithMilkAndSugar
coffeeWithMilkAndSugar	confirmSelection	MakeCoffee	idle

Table 2
State machine of the vending machine example

3.2. CppFSM

For creating finite state machines in C++, Philips IGT used the CppFSM framework. Listing 1 shows how to create a FSM in the CppFSM framework. It is a fragment of the state machine in Table 2. The

“addTransition” method adds transitions to the FSM in the following way:

1. First parameter is the current state. A state enumeration is used, but not shown in the listing.
2. Second parameter is the next state. The same state enumeration is used.
3. Third parameter is the input event. An input enumeration is used, but not shown in the listing.
4. Fourth parameter is the output event. This is a reference to a method.

Observe that a transition can only trigger one output event. The resulting state machine is input enabled which means that any event is accepted in any state; when no transition specified for a certain input event in a certain state, then there will be no output event, but also no assert will be called. Note that a CppFSM program can be seen as a component which has a provided interface with the input events and a required interface with the output events.

Listing 1: Vending machine FSM in C++

```
void createVendingFsm()
{
    mFsm.addTransition(idle,paid,insertTwoEuroCoin,&Fsm::displaySelectBeverage);
    mFsm.addTransition(paid,paid,insertTwoEuroCoin,&Fsm::returnCoin);
    mFsm.addTransition(paid,idle,cancel,&Fsm::returnCoin);
    mFsm.addTransition(paid,coffee,selectCoffee,&Fsm::displayCoffeeSelected);
...
    mFsm.addTransition(coffee,coffee,insertTwoEuroCoin,&Fsm::returnCoin);
    mFsm.addTransition(coffee,idle,cancel,&Fsm::returnCoin);
    mFsm.addTransition(coffee,coffeeWithMilk,selectMilk,&Fsm::displayMilkSelected);
    mFsm.addTransition(coffee,coffeeWithSugar,selectSugar,&Fsm::displaySugarSelected);

...
}
```

Listing 2 shows how the FSM is used. Transitions can be triggered by calling the “makeTransition” method. The current state can be queried for with the “getState” method. It returns the state enumeration value. An example of an output event is “displaySelectBeverage”. This method is registered as a function pointer in Listing 1. On a transition, the CppFSM framework can invoke this method.

Listing 2: Usage of FSM in C++

```
mFsm.makeTransition(FsmEvent::insertTwoEuroCoin);
mFsm.getState();

void Fsm::displaySelectBeverage() {
    // Implementation to display select beverage.
}
```

3.3. Dezyne

Dezyne[1] takes a component-based perspective on software. Every component typically has a provided interface and one or more required interfaces. There are two types of models:

- Interface model containing the signature and a behavioral protocol. The behavior protocol is written as a FSM which describes the allowed sequences of method calls.
- Component model describing component behavior, including usage of interface models. This behavior is also written as a FSM which describes what needs to be done when method calls happen on the provided and required interfaces.

Dezyne implements a run-to-completion semantics. This means that a component does not accept new method calls on its provided interface until the current method call has returned. In addition a

[1] https://gitlab.com/dezyne

component that uses the provided interface is blocked until the method call on the required interface returns [32].

With the Dezyne tool, interface and component models can be formally verified. This verification uses the mCRL2 model checker [12] which checks:

- Whether deadlock and livelock states are absent in interface and component models.
- Whether component models are deterministic. For interface models it is allowed to describe non-deterministic behavior.
- Whether a component is a refinement –using the failures divergence relation [33]– of its provided interface.
- Whether a component does not violate the behavioral protocol of its required interfaces.

From a Dezyne model, a mCRL2 model and C++ code can be generated [34]. Verum[2], the company that implements the Dezyne tooling, guarantees that then the C++ code and the mCRL2 model are semantically equivalent [14].

Typically, one first verifies a Dezyne model and if all checks pass, C++ code can be generated. With bindings the generated C++ code can easily be integrated into handwritten C++ code.

mCRL2 is used in the background and is hidden from the typical user. However, the tool can also store the mCRL2 model for the user to allow inspection or advanced checks.

4. Approach Applied to Industrial Case

In this section, we describe how we refine our general approach to a number of steps that transform CppFSM code to Dezyne. This also include a refactoring of the manually written code that interfaces with CppFSM to code that interfaces with Dezyne generated code.

Figure 3 depicts our detailed transformation approach for the Philips IGT case. The left code file depicts the original situation before transformation. We distinguish three types of code in a single code file:

- At the bottom, CppFSM code that implements a finite state machine.
- In the middle, code interfacing with the CppFSM code; it either calls FSM code or is called by the CppFSM code.
- At the top, all other code. This code interacts with the interfacing code.

On the right, we depict the situation after the transformation where the CppFSM code is removed. There is a new code file which includes the code that is generated from a Dezyne model.

We have automated the refactoring using the following eight steps:

1. We use a static technique to parse the code file and extract the Abstract Syntax Tree (AST).
2. From the AST, the CppFSM part is used to generate a Dezyne model.
3. Generate a mCRL2 model from the Dezyne model.
4. Use a dynamic technique to learn the behavior of the CppFSM code. The output is a Dot file. Dot is a language to describe nodes, edges and graphs[3].
5. Convert the Dot file to a mCRL2 model.
6. Compare both mCRL2 models using an equivalence checker.

When both models are different, fix the issue. Otherwise, continue with:

7. The Dezyne model is used to generate a new code file that contains the FSM.
8. We use the AST to refactor the code that interface with CppFSM to interface with the newly Dezyne generated code.

[2] https://www.verum.com/DiscoverDezyne
[3] https://graphviz.org/doc/info/lang.html

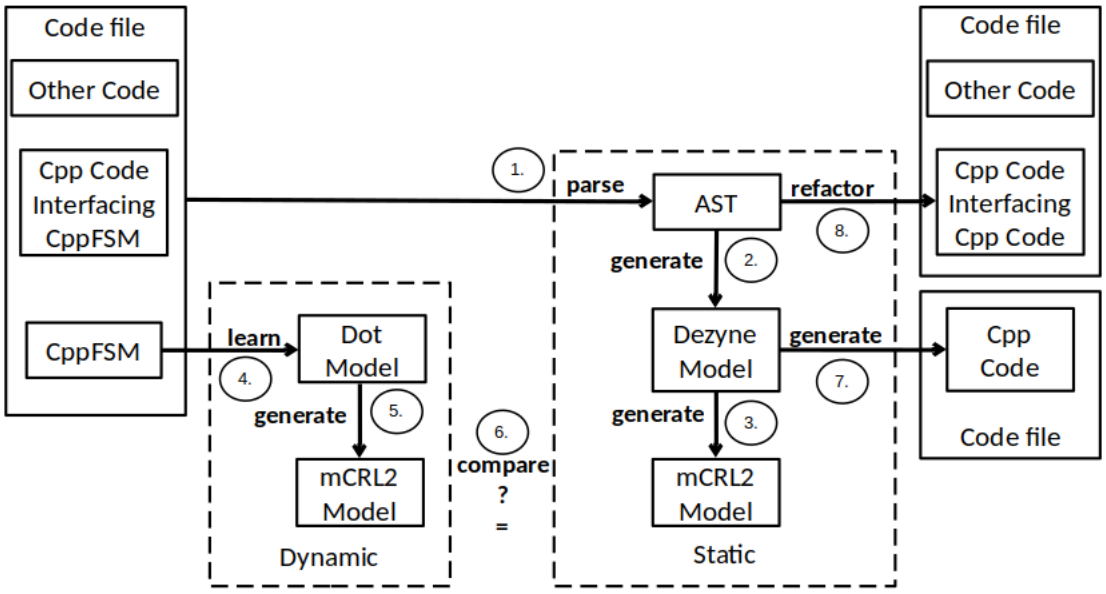

Figure 3: Combining static and dynamic techniques for refactoring CppFSMs

All steps are fully automated except for *Step 8* where some manual code changes might be required.

For the conversions in *Steps 1, 2, 5 & 8*, we use the Rascal metaprogramming language. We use Rascal in *Step 1* to generate the Dezyne model because if we would use a dynamic technique we would loose the state names. Hence with Rascal we can generate a Dezyne model with the state names defined in the CppFSM code. We use a state of art model learning algorithm L# [9] in *Step 4* to execute the legacy CppFSM code and observe its behavior as a black box. Next we compare the model extracted with the static technique with the model build by the dynamic technique.

The steps of our approach are explained in more detail in the subsequent sections. Section 4.1 describes the transformation from CppFSM to Dezyne. The learning setup of *Steps 4 & 5* is described in Section 4.2. The conversion of a learned model to a model that we can compare for equivalence is described in Section 4.3. Given two mCRL2 models, we describe *Step 6* in which we compare the two models in Section 4.4. The last step to refactor the code that interfaces with CppFSM to make it use of the newly generated Dezyne C++ code is explained in Section 4.5.

4.1. From CppFSM to Dezyne

Figure 3 *Steps 1, 2 & 7* depict the workflow that we follow to get from a CppFSM in C++ to a new C++ implementation generated by Dezyne. We use ClaiR, the C++ front end of Rascal to parse the C++ file with CppFSM.

Listing 1 from the case description in Section 3 is the input for the Rascal script. The output is Listing 3 which shows the resulting Dezyne model of the vending machine's state machine. As explained in Section 2, the component model imports two interface models: one is the provided interface and the other one the required interface. The states are defined in the "State" enumeration. The "state" variable of type "State" is initialized to the "idle" state. Next we see blocks between curly brackets with a state guard. The model should be read as follows:

- If in the "idle" state "insertTwoEuroCoin" is called, then "displaySelectBeverage" is shown and the next state is "paid".
- If in the "idle" state "cancel" is called, then it is accepted and the state remains "idle".

Note that the method "getState" returns the current state, but the method is not always needed. For this reason, we made the generation of this method optional. For this transformation, we used the Rascal code from Appendix A.

Listing 3: Component model of vending machine in Dezyne

```
import IVendingProvided.dzn;
import IVendingRequired.dzn;
component ExampleFSM {
    provides IVendingProvided iProvided;
    requires IVendingRequired iRequired;
    behaviour {
        enum State {idle,paid,coffee,coffeeWithSugar, coffeeWithMilk,coffeeWithMilkAndSugar};
        State state = State.idle;
        [state.idle] {
            on iProvided.insertTwoEuroCoin():{iRequired.displaySelectBeverage(); state = State.paid;}
            on iProvided.cancel(): ;
            on iProvided.selectCoffee(): ;
            on iProvided.selectMilk(): ;
            on iProvided.selectSugar(): ;
            on iProvided.confirmSelection(): ;
            on iProvided.getState(): reply(IProvided.State.idle);
        }
        ...
        [state.coffee] {
            on iProvided.insertTwoEuroCoin():{iRequired.returnCoin(); state = State.coffee;}
            ...
            on iProvided.getState(): reply(IProvided.State.coffee);
        }
        ...
    }
}
```

4.2. From CppFSM to Model Learning Setup

Figure 3 *Steps 4 & 5* depict the workflow that we follow to get from a CppFSM in C++ to a new mCRL2 model. Rascal is used to automatically generate a setup such that the behavior of the SUL can be learned. AML is utilized to acquire a Dot model and Rascal is used to transform the Dot model into a mCRL2 model. In Appenix B Listing 10, we show the generated learning setup in C++ code. When compiled, this code results in a console application. Standard input and output are used to connect the console application to the learner. The Rascal code in Appendix C Listing 11 prints the contents of the console application. This includes the SUL and the adapter. Listing 12 is used to generate the batch file to call the model learner with the right SUL and its input events (i.e. the input alphabet).

4.3. From Dot to mCRL2

Model learning results in a model in the Dot language. To allow equivalence checking –as described in Section 2– we need an mCRL2 model. Hence, we need to convert the Dot model to an mCRL2 model, according to Figure 3 *Step 5*. Listing 4 is an example of a Dot model as produced by the learner when learning the vending machine. Observe that the states are numbered from "s0" to "s7". For example, line 5 of Listing 4 is a transition from state "s0" to state "s1". In the learning process, the state names are lost because they are not externally visible.

Listing 4: Vending machine in Dot

```
digraph g {
    s0 [shape = "circle" label="s0"];
...
    s7 [shape = "circle" label="s0"];
    s0 -> s1 [label="insertTwoEuroCoin / displaySelectBeverage"];
    s0 -> s0 [label="cancel / self"];
    s0 -> s0 [label="selectCoffee / self"];
    s0 -> s0 [label="selectMilk / self"];
    s0 -> s0 [label="selectSugar / self"];
```

```
    s0 -> s0 [label="confirmSelection / self"];
...
    s4 -> s4 [label="insertTwoEuroCoin / returnCoin"];
    s4 -> s0 [label="cancel / returnCoin"];
    s4 -> s4 [label="selectCoffee / self"];
    s4 -> s6 [label="selectMilk / diplayMilkSelected"];
    s4 -> s5 [label="selectSugar / diplaySugarSelected"];
    s4 -> s0 [label="confirmSelection / makeCoffee"];
...
    __start0 [label="" shape="none" width="0" height="0"];
    __start0 -> s0;
}
```

Appendix D Listing 13 presents the result of translating a Dot model into an mCRL2 model. Listing 14 shows the Rascal script that performs the translation. Listing 15 also shows the Dot grammar in Rascal used by the script.

4.4. Equivalence Checking

In Figure 3 *Step 6*, there are two mCRL2 models; one acquired by a dynamic technique (AML) and one generated by a static technique (metaprogramming). In this step, we compare the behavior of the two models using mCRL2. If the equivalence check fails, the checker provides a counter example that leads to a discrepancy in the models. To resolve the issue, several aspects have to be investigated. One possibility is that the dynamic technique learned an incomplete model; then the counter example can be used to continue the learning process. An alternative is that one of the generators in the static approach is incorrect, for instance, because of a mismatch in the understanding of the semantics of the programming language. After resolving the discrepancy, the equivalence checker needs to be rerun on the updated model because there can be more than one discrepancy to resolve. If the equivalence check passes, the Dezyne tool can be utilized to generate new C++ code (*Step 7*).

4.5. Refactor Interfacing C++ Code

Figure 3 *Step 8* depicts the workflow for refactoring a code file. The code that interfaces with the CppFSM framework needs to be replaced by code that interfaces with Dezyne generated C++ code. Calls to the CppFSM framework need to be replaced by calls to Dezyne generated C++ code. Moreover, we have to create the bindings such that Dezyne generated code can invoke the manually written output event methods. In addition, the CppFSM itself is no longer needed, because we use Dezyne generated code, and needs to be removed from the code file.

Section 3 explained how CppFSM code needs to be used. In Listing 1 a vending machine FSM is shown. This code is no longer required. We replace it with bindings to output events (i.e. actions) which is required for connecting Dezyne generated code to manually written action methods. The new bindings are shown in Listing 5.

Listing 5: Vending machine bindings for Dezyne in C++

```
void createVendingFsm()
{
    mDezyneComposition.iRequired.in.displaySelectBeverage =
        std::bind(&Fsm::displaySelectBeverage, this);
    mDezyneComposition.iRequired.in.returnCoin =
        std::bind(&Fsm::returnCoin, this);
...
    mDezyneComposition.iRequired.in.displaySugarSelected =
        std::bind(&Fsm::displaySugarSelected, this);
}
```

In Listing 2, the “makeTransition” method and “insertTwoEuroCoin” input event is used to initiate a transition. This is replaced by “insertTwoEuroCoin” on the “iProvided” interface, see Listing 6. The Rascal script for the described refactoring is provided in Appendix E Listing 16.

Listing 6: Usage of Dezyne generated code in C++

```
mDezyneComposition.iProvided.in.insertTwoEuroCoin();
```

5. Results

We applied the described approach to CppFSM code from Philips IGT. To our surprise, we observed that in *Step 6* the mCRL2 models were not equivalent. When inspecting the learned model with a Dot viewer, we immediately realized that there was a semantic difference between the learned model and the generated Dezyne model. The learned model was input enabled, i.e., in all states all input events are allowed while the generated Dezyne model was not input enabled. We discovered that there were two versions of CppFSM: one for testing and one for in-product code. The testing flavor asserts on input events in states that do not have an explicit transition defined by "makeTransition". The product flavor, which was used in the Philips code, is input-enabled and accepts all inputs in all states. Hence, we had to make the generated Dezyne model also input enabled. We did this by creating the "makeStateMachineInputEnabled" method, see Listing 8 in Appendix A. After adding this method in the Dezyne model generator, the equivalence check holds.

After resolving this issue, we have applied the described approach to 14 Philips IGT cases. Table 3 depicts per case the number of states and transitions in both Dezyne generated and model learned mCRL2 models. In the table, we compare the mCRL2 models as introduced in Section 4. Using other models than the mCRL2 models, i.e., the Dezyne or Dot models would give different numbers for states and transitions. For comparison, we transform the mCRL2 models to Labelled Transition Systems (LTSs). In this way, the comparison only checks input/output relations.

	Statically obtained models		Dynamically acquired models	
Case	# States	# Transitions	# States	# Transitions
1	63	153	63	153
2	88	376	88	376
3	**42**	**70**	**6**	**10**
4	**84**	**182**	**79**	**170**
5	27	47	27	47
6	56	147	56	147
7	32	62	32	62
8	69	219	69	219
9	13	17	13	17
10	67	166	67	166
11	43	85	43	85
12	19	28	19	28
13	25	37	25	37
14	24	36	24	36

Table 3
Comparison of statically and dynamically acquired mCRL2 models

For all the refactored CppFSMs, the equivalence check holds which gave us a lot of confidence that refactoring was successful. Intuitively, one would expect that the equivalence check only holds when the number of states and transitions of the two models are equal. But in cases 3 and 4, we observed that the number of states and transitions is much lower in the dynamically learned model than in the statically obtained Dezyne model while the equivalence check holds. The reason is that the CppFSM did not have output events in these cases. This raises the question whether the Dezyne model can be simplified to fewer states and some transitions. When we investigated this further, we found out that in these two cases other code polls the "getState" method from Listing 2 to check the state of the FSM and makes different choices based on the returned state. To not break the manually written code, we added

the possibility to generate a "getState" in the Dezyne model and we manually refactored the usage of the "getState" from CppFSM to the new Dezyne model variant in the interfacing code.

In some cases, "makeTransition" from Listing 2 was embedded in another method. This method got the event enumeration as a parameter value. Our automated refactoring presented in Section 4.5 did not work in this case. Also in this case, we manually made the required changes.

6. Concluding Remarks

In this paper, we propose an approach to refactor Finite State Machines (FSMs) using a combination of static and dynamic techniques. This involves utilizing metaprogramming to generate a new FSM implementation from a source file containing a legacy FSM implementation. To increase confidence in the preserved semantics, we employ learning methods and equivalence checking. In this section, we describe limitations of our approach such as pitfalls and scalability in Section 6.1. Next we discuss alternative approaches in Section 6.2. Finally, in Section 6.3 we describe some ideas for future work.

6.1. Limitations

We refactored the C++ code that interfaces with CppFSM to interface with the code generated by Dezyne, but it is not included in the learned model. Consequently, this weakness requires reliance on existing regression test suites for verification of correctness. However, since we have automated the refactoring process for the interfacing code, our confidence in its correctness is higher than with manual alternatives.

6.1.1. Pitfalls

The main pitfalls concern the application of AML in an industrial setting. For instance, Omar et al. [35] have identified pitfalls while performing AML at Philips. Aslam [36] applied AML at ASML. In their setup, the adapter and the SUL are placed in separate executables and the adapter connects to the SUL using a TCP/IP socket connection. This setup is suboptimal because it is very slow to send and receive messages over a TCP/IP stack, to let the Operating System (OS) stop and start an executable for a reset, and to reestablish the TCP/IP connection. To mitigate this performance issue, we run the adapter and SUL in a single process, as shown in Appendix B Listing 10.

Omar et al. have identified that adapters could be generated. By generating the adapter, there are fewer opportunities to learn faulty models due to issues with the learning setup compared to the manual creation of adapters. Aslam et al. created a generator for the learning setup and we have also automated this process as described in Section 4.2.

Both identified issues with scalability, which we address in the next section.

6.1.2. Scalability

State space explosion is a valid concern for both model learning and model checking [37]. Because of the time required to query and reset the SUL, AML will suffer from state space explosion before equivalence checking runs into the state space explosion problem. Hence, as shown by Aslam, it might be the case that complex models cannot be learned in a reasonable amount of time due to the limited ability to capture far-output-distinction behavior. To find far-output-distinction behavior, long EQs are required that distinguish the hypothesized model from the SUL's behavior. For instance, if "input1" produces an "output1". However, after calling "input1" one hundred times it no longer produces "output1", but "output2" [36]. By manual inspections of our CppFSMs before refactoring we know that such a pattern was not present in our industrial case.

In addition, we do not expect scalability issues with metaprogramming because creating an AST and perform functions using the AST is done in just a few seconds.

6.2. Alternative Approaches

In this paper, we employed metaprogramming as a static technique and AML as a dynamic technique. However, other alternatives are possible.

Instead of using metaprogramming, we could have utilized a Language WorkBench (LWB) to create a Domain Specific Language (DSL) [2]. The DSL should be capable of parsing the "addTransition" method lines of Listing 1. The DSL can be employed to generate mCRL2 models. In fact, for this approach and the one used in the paper, the same mCRL2 model code generator could be reused. The only difference is in *Step 8*, as a DSL cannot refactor the C++ code file to use the generated C++ code with the Dezyne tool. Alternatively, manual refactoring can be performed.

An alternative dynamic approach is to execute the code using test cases. Typically, testing such FSMs will be done using unit testing. However, legacy code is characterized by limited code coverage of the tests [38]. To mitigate this issue, techniques can be employed to improve existing unit test suites by applying, e.g., mutation testing [39] or test amplification [40]. Downside of these techniques is that most implementations only support Java and not C++ which is our target language. The benefit of using AML is that it is programming language agnostic.

6.3. Future Work

For this case, we manually checked that far-output-distinction behavior did not occur. Alternatively, we could have automated this check using a small metaprogram with, i.e., Rascal. Or we could have used a code coverage tool to check if all code is executed by AML.

There is a limitation where the refactored interfacing code is not learned due to its separation from the FSM implementation. For the correctness of this refactoring, we relied on the existing regression test suites. In the future, we want to investigate if we could include this code in the learned model.

References

[1] H. van Vliet, Software engineering: principles and practice, volume 13, John Wiley & Sons Hoboken, NJ, 2008.

[2] M. Fowler, Domain-specific languages, Pearson Education, 2010.

[3] P. Klint, T. Van der Storm, J. Vinju, RASCAL: A domain specific language for source code analysis and manipulation, Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE, 2009, pp. 168–177. doi:10.1109/SCAM.2009.28.

[4] P. Tonella, Reverse engineering of object oriented code, in: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., IEEE, 2005, pp. 724–725.

[5] M. Von der Beeck, A comparison of statecharts variants, in: Formal Techniques in Real-Time and Fault-Tolerant Systems: Third International Symposium Organized Jointly with the Working Group Provably Correct Systems—ProCoS Lübeck, Germany, September 19–23, 1994 Proceedings 3, Citeseer, 1994, pp. 128–148.

[6] F. Vaandrager, Model learning, Communications of the ACM 60 (2017) 86–95.

[7] D. Angluin, Learning regular sets from queries and counterexamples, Information and computation 75 (1987) 87–106.

[8] M. Isberner, F. Howar, B. Steffen, The TTT algorithm: a redundancy-free approach to active automata learning, in: Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5, Springer, 2014, pp. 307–322.

[9] F. Vaandrager, B. Garhewal, J. Rot, T. Wißmann, A new approach for active automata learning based on apartness, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2022, pp. 223–243.

[10] F. B. Khendek, S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, Test selection based on finite state models, IEEE Transactions on software engineering 17 (1991) 10–1109.

[11] M. Soucha, K. Bogdanov, State identification sequences from the splitting tree, Information and Software Technology 123 (2020) 106297.
[12] J. F. Groote, M. Mousavi, Modelling and analysis of communicating systems, MIT press, 2014.
[13] J. Engelfriet, Determinancy→(observation equivalence= trace equivalence), Theoretical Computer Science 36 (1985) 21–25.
[14] R. van Beusekom, B. de Jonge, P. Hoogendijk, J. Nieuwenhuizen, Dezyne: Paving the way to practical formal software engineering, arXiv preprint arXiv:2108.02962 (2021).
[15] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.
[16] J. Ivers, I. Ozkaya, R. Nord, C. Seifried, Next generation automated software evolution refactoring at scale, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1521–1524.
[17] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, N. Amaral, B. Y. Chang, S. Guyer, U. Khedker, A. Møller, D. Vardoulakis, In defense of soundiness: A manifesto, Communications of the ACM 58 (2015) 44–46.
[18] M. Van den Brand, A. Van Deursen, J. Heering, H. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, J. Visser, The ASF+SDF meta-environment: A component-based language development environment, Electron. Notes Theor. Comput. Sci. 44 (2001) 3–8. doi:10.1016/S1571-0661(04)80917-4.
[19] E. Visser, Program transformation with Stratego/XT, in: C. Lengauer, D. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation: International Seminar, Springer, 2004, pp. 216–238. doi:10.1007/978-3-540-25935-0_13.
[20] J. Cordy, T. Dean, A. Malton, K. Schneider, Source transformation in software engineering using the TXL transformation system, Inf. Softw. Technol. 44 (2002) 827–837. doi:10.1016/S0950-5849(02)00104-0.
[21] O. Bagge, K. Kalleberg, M. Haveraaen, E. Visser, Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs, Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation, IEEE, 2003, pp. 65–74. doi:10.1109/SCAM.2003.1238032.
[22] I. Baxter, C. Pidgeon, M. Mehlich, DMS®: program transformations for practical scalable software evolution, Proceedings of the 26th International Conference on Software Engineering, IEEE, 2004, pp. 625–634. doi:10.1109/ICSE.2004.1317484.
[23] D. Waddington, B. Yao, High-fidelity C/C++ code transformation, Electron. Notes Theor. Comput. Sci. 141 (2005) 35 – 56. doi:10.1016/j.entcs.2005.04.037.
[24] H. Wright, D. Jasper, M. Klimek, C. Carruth, Z. Wan, Large-scale automated refactoring using ClangMR, Proceedings of the 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 548–551.
[25] A. Mooij, J. Ketema, S. Klusener, M. Schuts, Reducing code complexity through code refactoring and model-based rejuvenation, Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, IEEE, 2020, pp. 617–621. doi:10.1109/SANER48275.2020.9054823.
[26] M. Schuts, R. Aarssen, P. Tielemans, J. Vinju, Large-scale semi-automated migration of legacy C/C++ test code, Software: Practice and Experience 52 (2022) 1543–1580.
[27] R. Aarssen, cwi-swat/clair: v0.1.0 (2017). doi:10.5281/zenodo.891122.
[28] S. Ali, H. Sun, Y. Zhao, Model learning: a survey of foundations, tools and applications, Frontiers of Computer Science 15 (2021).
[29] W. Smeenk, J. Moerman, F. Vaandrager, D. Jansen, Applying automata learning to embedded control software, in: International Conference on Formal Engineering Methods, Springer, 2015, pp. 67–83.
[30] M. Schuts, J. Hooman, F. Vaandrager, Refactoring of legacy software using model learning and equivalence checking: an industrial experience report, in: International Conference on Integrated Formal Methods, Springer, 2016, pp. 311–325.
[31] M. Schuts, J. Hooman, P. Tielemans, Industrial experience with the migration of legacy models

using a DSL, in: Proceedings of the Real World Domain Specific Languages Workshop 2018, 2018, pp. 1–10.
[32] J. Hooman, R. Huis in't Veld, M. Schuts, Experiences with a compositional model checker in the healthcare domain, in: International Symposium on Foundations of Health Informatics Engineering and Systems, Springer, 2011, pp. 93–110.
[33] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. W. Roscoe, FDR3—a modern refinement checker for CSP, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2014, pp. 187–201.
[34] R. v. Beusekom, J. F. Groote, P. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, T. A. Willemse, Formalising the dezyne modelling language in mCRL2, in: Critical Systems: Formal Methods and Automated Verification, Springer, 2017, pp. 217–233.
[35] O. al Duhaiby, A. Mooij, H. van Wezep, J. F. Groote, Pitfalls in applying model learning to industrial legacy software, in: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV 8, Springer, 2018, pp. 121–138.
[36] K. Aslam, Deriving behavioral specifications of industrial software components, PhD thesis, TU/e, 2021.
[37] M. Schuts, J. Hooman, Towards an industrial stateful software rejuvenation toolchain using model learning, in: Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2023, pp. 15–31.
[38] T. Weigert, A. Kolchin, S. Potiyenko, O. Gurenko, A. van den Berg, V. Banas, R. Chetvertak, R. Yagodka, V. Volkov, Generating test suites to validate legacy systems, in: System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0: 11th International Conference, SAM 2019, Munich, Germany, September 16–17, 2019, Proceedings 11, Springer, 2019, pp. 3–23.
[39] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: an analysis and survey, in: Advances in computers, volume 112, Elsevier, 2019, pp. 275–378.
[40] C. Brandt, A. Zaidman, Developer-centric test amplification: The interplay between automatic generation human exploration, Empirical Software Engineering 27 (2022).

A. From CppFSM To Dezyne

Listing 7: Parse C++

```
private lrel[str, str, str, str] parseCppFsmTable(f) {
    ast = parseCpp(f);
    lrel[str, str, str, str] sm = [];
    visit (ast) {
        case \functionCall(\fieldReference(_, \name("addTransition")),
               [\idExpression(\name(curState)), \idExpression(\name(nextState))
               \idExpression(\name(event)), actionField, *_]): {
            action = "nullPointer";
            visit (actionField) {
                case /idExpression(\qualifiedName(_, \name(a))):
                    action = a;
            }
            sm += <curState, nextState, event, action>;
        }
    }
    return sm;
}
```

Listing 8: Make FSM input enabled

```
private lrel[str, str, str, str] makeInputEnabled(sm) {
       lrel[str, str, str, str] rVal = sm;
```

```
        allEvents = { e | <c, n, e, a> <- sm };
        allStates = { c | <c, n, e, a> <- sm } + { n | <c, n, e, a> <- sm};
        for (state <- allStates) {
               transForEvents = { e | <c, n, e, a> <- sm, c == state };
               noTransForEvents = allEvents - transForEvents;
               for (noTransForEvent <- noTransForEvents) {
                      rVal += <state, state, noTransForEvent, "nullptr">; // add self-transition
               }
        }
        return rVal;
}
```

Listing 9: Generate Dezyne component model

```
private str printDezyneComponent(sm,generateGetState) {
    rVal = "";
    states = { c | <c, n, e, a> <- sm } + { n | <c, n, e, a> <- sm };
    rVal += "import IProvided.dzn;
            'import IRequired.dzn;
            'component genFSM {
            '    provides IProvided iProvided;
            '    requires IRequired iRequired;
            '    behaviour {
            '";
    strippedStates = replaceAll("<states>", "\"", "");
    rVal += "       enum State <strippedStates>;
            '       State state = State.<head(toList(states))>;
            '";
    for (s <- states) {
        rVal += "       [state.<s>] {\n";
        trans = { <n, e, a> | <c, n, e, a> <- sm, c == s };
        for (<n, e, a> <- trans) {
            rVal += "           on iProvided.<e>(): {
                    '               state = State.<n>;
                    '           }
                    '";
        }
        if (generateGetState) {
            rVal += "           on iProvided.getState(): {
                    '               reply(IProvided.eState.<s>);
                    '           }
                    '";
        }
        rVal += "       }\n";
    }
    rVal += "   }
            '}
            '";
    return rVal;
}
```

B. Model Learning Setup

Listing 10: SUL and Adapter in C++

```
class SulFsmClient {
private:
    typedef Fsm<SulFsmClient, SulFsmEvent, SulFsmState> SulFsm;
public:
...
    SulFsmClient() :mSulFsm(*this, idle, card, nullptr) {
        createFsm();
```

```
    }
    void displaySelectBeverage() {
        mOutput = "displaySelectBeverage";
    }
...
    void selectSugar() {
        mOutput = "selectSugar";
    }
    void step(string symb) {
        mOutput = "self";
        if (symb == "insertTwoEuroCoin") {
            mSulFsm.makeTransition(SulFsmClient::insertTwoEuroCoin);
        }
...
        if (symb == "selectSugar") {
            mSulFsm.makeTransition(SulFsmClient::selectSugar);
        }
        cout << mOutput << endl;
    }
private:
    void createFsm()
    {
        mFsm.addTransition(idle,paid,insertTwoEuroCoin,&Fsm::displaySelectBeverage);
    ...
        mFsm.addTransition(coffee,coffeeWithSugar,selectSugar,&Fsm::displaySugarSelected);
    ...
    }
    SulFsm mSulFsm;
    string mOutput;
};
int main(void) {
    SulFsmClient* mFsmClient = new SulFsmClient();
    while (true) {
        string symb = "";
        getline(cin >> ws, symb);
        if (symb.length() == 0) {
            getline(cin >> ws, symb);
        }
        if (symb.compare("RESET") == 0) {
            delete mFsmClient;
            mFsmClient = new SulFsmClient();
        } else {
            mFsmClient->step(symb);
        }
    }
    return 0;
}
```

C. From CppFSM To Model Learning Setup

Listing 11: Parse C++ and generate learning setup

```
private str printLearnSetup(sm) {
    rVal = "";
    rVal +=    "#include \<string\>
           '#include \<iostream\>
           '#include \"GenFsm.hpp\"
           'using namespace std;
           'namespace gen {
           '    class SulFsmClient {
           '    public:
           '        enum SulFsmEvent {\n";
```

```
    events = { e | <c, n, e, a> <- sm };
    for (event <- events) { rVal += "        <event>,\n"; }
   rVal += "       card };
          '       enum SulFsmState {\n";
    states = { c | <c, n, e, a> <- sm } + { n | <c, n, e, a> <- sm };
    for (state <- states) { rVal += "        <state>,\n"; }
    rVal += "       };
           '    private:
           '        typedef Fsm\<SulFsmClient, SulFsmEvent, SulFsmState\> SulFsm;
           '    public:
           '        SulFsmClient() :mSulFsm(*this, <head(toList(states))>, card, nullptr) {
           '            createFsm();
           '        }\n";
    calls = { a | <c, n, e, a> <- sm };
    for (call <- calls) {
        rVal += "      void <call>() { mOutput = \"<call>\"; }\n";
    }
    rVal += "      void step(string symb) { mOutput = \"self\";\n";
    for (event <- events) {
        rVal += "          if (symb == \"<event>\") {
               '        mSulFsm.makeTransition(SulFsmClient::<event>);
               '            }\n";
    }
    rVal += "          cout \<\< mOutput \<\< endl;
           '        }
           '    private:
           '        void createFsm() {\n";
    for (<c, n, e, a> <- sm) {
        rVal += "           mSulFsm.addTransition(<c>, <n>, <e>, &SulFsmClient::<a>);\n";
    }
    rVal += "       }
           '    private:
           '        SulFsm mSulFsm;
           '        string mOutput;
           '};
           '} // namespace
           'using namespace gen;
           'int main(void) {
           '    SulFsmClient* mFsmClient = new SulFsmClient();
           '    while (true) {
           '        string symb = \"\";
           '        getline(cin \>\> ws, symb);
           '        if (symb.length() == 0) {
           '            getline(cin \>\> ws, symb);
           '        }
           '        if (symb.compare(\"RESET\") == 0) {
           '            delete mFsmClient;
           '            mFsmClient = new SulFsmClient();
           '        } else {
           '            mFsmClient-\>step(symb);
           '        }
           '    }
           '    return 0;
           '}\n";
    return rVal;
}
```

Listing 12: Parse C++ and generate learning batch file

```
private str printBatch(sm, oFile) {
    rVal = "";
    splitted = split("/", "<oFile>");
    bat = last(splitted);
    exe = replaceAll(bat, "bat|", "exe");
```

```
    rVal += "cargo run --";
    events = { e | <c, n, e, a> <- sm };
    for (event <- events) {
        rVal += " -I <event>";
    }
    rVal += " -M experiments\\<exe>\n";
    return rVal;
}
```

D. From Dot To mCRL2

Listing 13: Vending machine in mCRL2

```
proc
genFSM'behavior (state': genFSM'State'enum) = genFSM's_ ();
genFSM's_(state': genFSM'State'enum) =
   genFSM'insertTwoEuroCoin_s0 () + ... + genFSM'insertTwoEuroCoin_s7 ()
 + genFSM'cancel_s0 () + ... + genFSM'cancel_s7 ()
...
;
genFSM'insertTwoEuroCoin_s0(state': genFSM'State'enum) = (state' == genFSM'State's0) ->
    iProvided'in (IProvided'action (IProvided'in'insertTwoEuroCoin)) .
        iRequired'in (IRequired'action (IRequired'in'displaySelectBeverage)) .
        iRequired'reply(IRequired'Void(void)) .
    genFSM'insertTwoEuroCoin_s0_(state' = genFSM'State's1);
        genFSM'insertTwoEuroCoin_s0_(state': genFSM'State'enum) =
        iProvided'reply (IProvided'Void(void)) .
        hide1 . hide2 .
        genFSM'behavior ();
genFSM'cancel_s0(state': genFSM'State'enum) = (state' == genFSM'State's0) ->
    iProvided'in (IProvided'action (IProvided'in'cancel)) .
    genFSM'cancel_s0_(state' = genFSM'State's0);
        genFSM'insertTwoEuroCoin_s0_(state': genFSM'State'enum) =
        iProvided'reply (IProvided'Void(void)) .
        hide1 . hide2 .
        genFSM'behavior ();
...
init hide({hide1, hide2}, genFSM'behavior (genFSM'State's0));
```

Listing 14: Parse Dot and generate mCRL2

```
private str printmCRL2(sm) {
    var rVal = "";
    states = { c | <c, n, e, a> <- sm } + { n | <c, n, e, a> <- sm };
    events = { e | <c, n, e, a> <- sm };
    actions = { a | <c, n, e, a> <- sm };
    rVal += "genFSM\'behavior (state\': genFSM\'State\'enum) = genFSM\'s_ ();
            'genFSM\'s_(state\': genFSM\'State\'enum) =\n";
    isFirst = true;
    for (e <- events) {
        for (s <- states) {
            rVal += isFirst ? " " :" + ";
            rVal += "genFSM\'<e>_<s> ()\n";
            isFirst = false;
        }
    }
    rVal += ";\n\n";
    for (event <- events) {
        for (<curState, nextState, event, action>
                <- { <c, n, e, a> | <c, n, e, a> <- sm, event == e }) {
            rVal += "genFSM\'<event>_<curState>(state\': genFSM\'State\'enum) =
                            (state\' == genFSM\'State\'<curState>) -\>
```

```
                '    iProvided\'in (IProvided\'action (IProvided\'in\'<event>)) .\n";
        if (action != "self") {
            rVal += "       iRequired\'in (IRequired\'action (IRequired\'in\'<action>)) .
                   '       iRequired\'reply(IRequired\'Void(void)) .\n";
        }
        rVal += "   genFSM\'<event>_<curState>_(state\' = genFSM\'State\'<nextState>);
               '       genFSM\'<event>_<curState>_(state\': genFSM\'State\'enum) =
               '       iProvided\'reply (IProvided\'Void(void)) .
               '       hide1 . hide2 .
               '       genFSM\'behavior ();\n";
    }
    }
    rVal += "\n\ninit hide({hide1, hide2}, genFSM\'behavior (genFSM\'State\'s0));\n\n";
    return rVal;
}
```

Listing 15: Dot grammar

```
layout Layout = WhitespaceAndComment* !>> [\ \t\n\r#];
lexical WhitespaceAndComment = [\ \t\n\r] | @category="Comment" "#" ![\n]* $;

start syntax Build = build: "digraph" "g" "{" State+ states Transition+ transitions
    Footer footer"}";
syntax State = state: Id name "[" "shape" "=" "\"circle\"" "label" "=" "\""
    Id text "\"" "];";
syntax Transition = transition: Id curState "-\>" Id nextState "[" "label" "=" "\""
    Id event "/" Id call "\"];";
syntax Footer = footer: "__start0" "[" "label" "=" "\"\"" "shape" "=" "\"none\""
    "width" "=" "\"0\"" "height" "=" "\"0\"" "];" "__start0" "-\>" "s0;";
lexical Id = ([a-zA-Z/.\-][a-zA-Z0-9_/.]* !>> [a-zA-Z0-9_/.]) \ Reserved;
keyword Reserved = "digraph" | "g" | "State" | "," | ";" | "=" | "{" | "}" | "[" | "]"
    | "on" | "state";
```

E. Replace Legacy C++ By Dezyne Generated C++

Listing 16: Replace CppFSM glue by Dezyne glue code

```
private lrel[int, int, str] refactorGenFSMTable(f) {
    lrel[int, int, str] changes = [];
    list[str] actions = [];
    ast = parseCpp(f);
    className = getClassName(ast);
    visit (ast) {
        case e:\expressionStatement(\functionCall(\fieldReference(_, \name("addTransition")),
               [\idExpression(\name(curState)), \idExpression(\name(nextState))
               \idExpression(\name(event)), actionField, *_])): {
            action = "nullPointer";
            visit (actionField) {
                case /idExpression(\qualifiedName(_, \name(a))): {
                    action = a;
                }
            }
            if (action != "nullPointer" && action notin actions) {
                changes += <e.src.offset, e.src.offset + e.src.length,
                    "mDezyneComposition.iRequired.in.<action> = std::bind(&<className>::<action>, this
                        );">;
                actions += action;
            } else {
                changes += <e.src.offset, e.src.offset + e.src.length, "">;
            }
        }
        case e:\expressionStatement(\functionCall(\fieldReference(_, \name("makeTransition")),
```

```
                [\idExpression(name(event))])): {
            changes += <e.src.offset, e.src.offset + e.src.length,
                "mDezyneComposition.iProvided.in.<camelCaseName(event)>();">;
        }
    }
    return changes;
}
```