Towards an Industrial Stateful Software Rejuvenation Toolchain using Model Learning

Mathijs Schuts
mathijs.schuts@ru.nl
Radboud University
Nijmegen, Netherlands

Jozef Hooman
jozef.hooman@tno.nl
TNO-ESI
Eindhoven, Netherlands

Abstract

We present our vision for creating an industrial legacy software rejuvenation toolchain. The goal is to semi automatically remove code smells from stateful software used in Cyber Physical Systems (CPS). Compared to existing tools that remove code smells, our toolchain can remove more than one type of code smell. Additionally, our approach supports multiple programming languages because we use abstract models obtained by means of model learning. Supporting more than one programming language is often lacking in state of art refactoring tools.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; *State based definitions.*

Keywords: model learning, model based development, software refactoring, software rejuvenation, state machine

ACM Reference Format:
Mathijs Schuts and Jozef Hooman. 2023. 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 (Onward! '23), October 25–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3622758.3622888

1 Introduction

The high tech industry creates Cyber Physical Systems (CPS) –such as cars, airplanes, trains, medical systems and industrial robots– that contain a lot of software to manage and control hardware. Often these embedded systems have software architectures in which many decades of development has been invested. Up to 80 percent of development cost is spend on maintaining embedded systems. Some software components can be characterized as legacy. Legacy components lack documentation and often the original developers have left the company. Moreover, obsolete tools, libraries and languages are used, and regression test coverage is very limited. Unfortunately, frequent updates and extensions of the software by a large number of software developers working under time pressure, inevitably leads to a lot of incidental complexity [28]. Such incidental complexity can be detected by so called code smells [12].

Legacy software contains a lot of accumulated value that cannot be thrown away to start from scratch. Starting from scratch would mean that scarce resources are used to make a new implementation of the system with the same behavior as the current system. These resources cannot be used to create new functionality and features for customers.

State of art automated refactoring tools typically focus on the removal of one type of code smell, e.g. code duplication. They also provide support for refactoring software written in a single programming language only. In addition, these tools are based on static analysis techniques [23]. In Section 2, we elaborate on state of art related work.

The goal of this paper is to propose a toolchain for the semi-automated refactoring of embedded software. In industrial practice, a complete rewrite of the code is almost never economically feasible. Moreover, because of the time pressure mentioned before, the set of tests is typically limited. Hence, it is difficult to show that a new implementation has the same behavior as the original system. We aim at automatic support of code refactorings where we have more confidence in the correctness of the code changes than by only running the existing regression test cases.

Our approach combines model learning, equivalence checking and metaprogramming technologies to remove incidental complexity by refactoring code and removing code smells. Removing code smells will improve the maintainability characteristics of an implementation [23].

In CPS, the behavior is often written in terms of state machines. For instance, a state machine can represent the states of a hardware component that is managed by software. Our proposed toolchain acquires a state machine model of the blackbox dynamic behavior of a complex software system. Model learning [32] is a technique to capture the behavior of an implementation in the form of a state machine. Because model learning is a blackbox technique, our proposed tool is not bound to a single programming language. It can –in contrast to the existing state of art tools– potentially refactor software written in any programming language.

Onward! '23, October 25–27, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0388-1/23/10.
https://doi.org/10.1145/3622758.3622888

Metaprogramming is used to generate a learning setup automatically, to connect the different technologies of our toolchain and to generate new code.

Learned models can be large when a learned implementation is large. To restructure the state machine, the toolchain supports the possibility to edit the learned model interactively with model based tools. The edited model can be checked for equivalence with the learned model using an equivalence checker. In this way it can be ensured that the behavior is preserved by the edits.

From the edited model, a new implementation can be generated. The generated software will be a formfit replacement of the original legacy software. Hence, all external interfaces will be respected. The newly generated software has the code smells removed.

We formulate the following requirements for the toolchain:

- **Req. 1** The toolchain shall be able to rejuvenate units of state-based CPS.
- **Req. 2** The toolchain shall support multiple programming languages.
- **Req. 3** The toolchain shall remove many code smells.
- **Req. 4** The toolchain shall provide a formfit replacement. Hence, the rejuvenated code has unchanged external interfaces and the behavior of the code is equivalent.

The toolchain consists of a novel combination of existing technologies. The toolchain combines model learning, equivalence checking and metaprogramming technologies. We intend to make the toolchain modular such that we can innovate the steps in the workflow by using new or improved tools, techniques or algorithms. We show that the individual technologies have already been successfully applied in industry. We reason that the combination of the technologies will implement the described requirements while existing tools do not.

The paper is organized as follows. In Section 2, we elaborate on state of art related work. Section 3 provides background information about the technologies and describes examples of their application in industry. We describe the refactoring workflow in Section 4, and in Section 5, we describe our plan for a first prototype of the toolchain. Section 6 discusses challenges and limitations. Section 7 contains concluding remarks.

2 Related Work

Requirements 3 & 4 are inspired by the book of Fowler [12] about refactoring. He defines refactoring as changing the internals of an unit (file) without changing its usage of its external interface while preserving its behavior. When the current test suites have insufficient coverage, the manual creation of test cases is advocated before the start of the refactoring. Moreover, Fowler describes how to manually remove code smells after the manually creation of additional test cases.

Lacerda et. al. [23] performed a tertiary Systematic Literature Review (SLR) –which is a review of secondary studies– about refactoring and code smells. The paper provides an extensive list of the type of code smells identified by multiple authors. They also list many tools that could be used to identify refactoring opportunities. There are only a few tools that can automate code smell refactorings. The studies they have included presented 24 different refactor automation tools. These tools only remove one or few code smells. In addition, the tools only support one programming language, i.e., Java or Erlang. The largest challenge for these tools is to preserve behavior. In sum, existing tools do not implement the *Reqs. 2-4* we identified in Section 1.

Integrated Development Environments (IDE) like Visual Studio[1], Visual Code[2] and Eclipse[3] provide support to refactor C/C++/C# code, for instance to rename methods and variables. However, these IDEs does not support the semi automated removal of code smells [23]. Hence, IDEs do not fulfil our requirements for a legacy software rejuvenation tool.

Agnihotri and Anyradha [3] conducted a secondary SLR about software metrics, code smells and refactoring techniques. In their paper, they conclude that most research on refactoring and code smells is conducted on Java based programs. Hardly any research is performed on the C, C++ and C# programming languages which are dominant in the CPS software domain which is our target, as mentioned in *Req. 1*.

Ivers et. al. [20] describe their vision on automated code refactoring tools. In their vision, support for automated refactoring should be generalized for multiple programming languages, like Java, C++ and C#, because of their –sometimes subtle– differences in semantics. This is in line with our *Req. 2*.

Meta programming can be applied to large scale automated refactoring of code. Limitation of this technique is that there must be regression test suites which provide enough confidence that the behavior is preserved as identified by [29]. So *Req. 4* is hard to guarantee for these kind of tools.

Concluding, our literature research did not reveal a tool that adheres to the requirements we described in Section 1.

3 Background of Technologies

We provide background information about the technologies we intent to use for the toolchain. Per technology, we also describe their application in industry.

3.1 Model Learning

Model learning is a technique to construct behavioral models from existing software. For this, the Minimal Adequate

[1] https://visualstudio.microsoft.com
[2] https://code.visualstudio.com
[3] https://www.eclipse.org

Teacher (MAT) framework as is shown in Figure 1 is used. The learner uses a teacher to acquire a model from the System Under Learning (SUL). The learner is instrumented with a set of possible inputs it can send to the teacher. When learning starts the learner sends sequences of inputs called Membership Queries (MQ) to the teacher. For every sequence of inputs, the teacher returns a sequence of outputs. Before a new input sequence is send, the SUL is reset to its initial state. After some MQs, the learner has "enough" information to build a hypothesis. The hypothesis is tested with an Equivalence Query (EQ). The teacher can either confirm the hypothesis –in this case the hypothesis is the final model– or otherwise it will return a counterexample. The counterexample is used by the learner to continue learning and create a new hypothesis. The resulting model is a (deterministic) Mealy state machine [6].

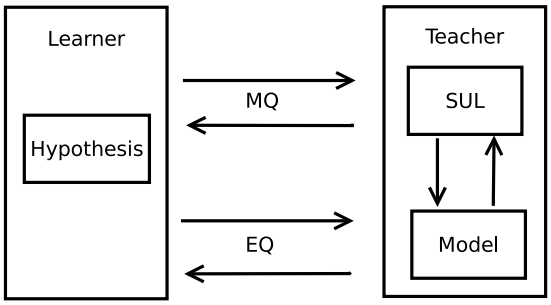

Figure 1. The MAT framework

Model learning has been applied on quite a number of 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 [5]. Also in industry model learning has been used. For instance, at Canon a controller of a high end printing copier of 410 states and 77 stimuli was learned [31].

Schuts et. al. [30] write about their experience with the application of model learning in the context of refactoring software. Model learning was used to check if a manual reimplementation was indeed a behavior preserving refactoring. Models were learned from the old and the new implementation. Next they used an equivalence checker to compare the two models. The equivalence checker found some discrepancies.

3.2 Equivalence Checking

To get confidence in the correctness after manual edits, we want to check equivalence of a model of the old implementation with a model of the new implementation. We can use a model checker to prove the equivalence of two models. In the domain of formal methods, different relations exist to compare two state machines where (weak) trace is the weakest and (strong) bisimulation is the strongest relation. When two Mealy state machines do not contain any silent transitions, trace equivalence and bisimulation equivalence coincide, and there is no difference between weak and strong equivalences [11]. Model checkers with support for strong bisimulation are, for instance, mCRL2 [16], CADP [15] and TAPAs [10].

3.3 Model Based Tools

The learned state machines can become quite large. To handle complexity, we want to decompose them into smaller state machines. Model based tools like Rhapsody [17] provide ways to handle complexity. In Rhapsody, state machines are designed using a graphical design window. To reduce complexity it is for instance possible to design a hierarchical state machine. Compared to a flat state machine, states are grouped in a hierarchical state machine. This reduces the complexity for a human reader. The way to implement a state machine is hidden by the Rhapsody run-time. From the graphical design a state machine implementation is automatically generated. An alternative is Visual State [35].

Another category are light-weight formal tools, like ASD, Cocotec[4] and Dezyne[5] [34]. The latter has a proprietary version and an open source version[6] These three tools have a DSL to describe state behavior. Internally, the tools convert a DSL instance to a formal model [9]. A model checker is applied to test the formal model for the absence of deadlock and livelock. It also checks if the implementation conforms to its interface specification. In addition, it checks if required components are used according their interface specification. This makes the approach compositional. If all checks pass, source code can be generated and integrated into the product.

Osaiweran et. al. calculated that this technology reduces the number of defects and increases productivity compared to manually written source code [25].

All metioned model based tools supports the generation of source code for multiple commonly used programming languages in the CPS domain.

3.4 Metaprogramming

For model learning, we need to connect the SUL to the learner. This needs to be done based on the external interfaces of the SUL. Metaprogramming languages can be used to generate an adapter to connect the learner to the SUL. An Abstract Syntax Tree (AST) of the external interfaces can be used for code generation. Examples of metaprogramming languages with C-style language support are Rascal [21], Proteus [2] and CodeBoost [7].

4 Proposed Toolchain

In this section, we describe how to combine the technologies of the previous section to create an embedded software

[4] https://cocotec.io/
[5] https://www.verum.com/
[6] https://github.com/dezyne

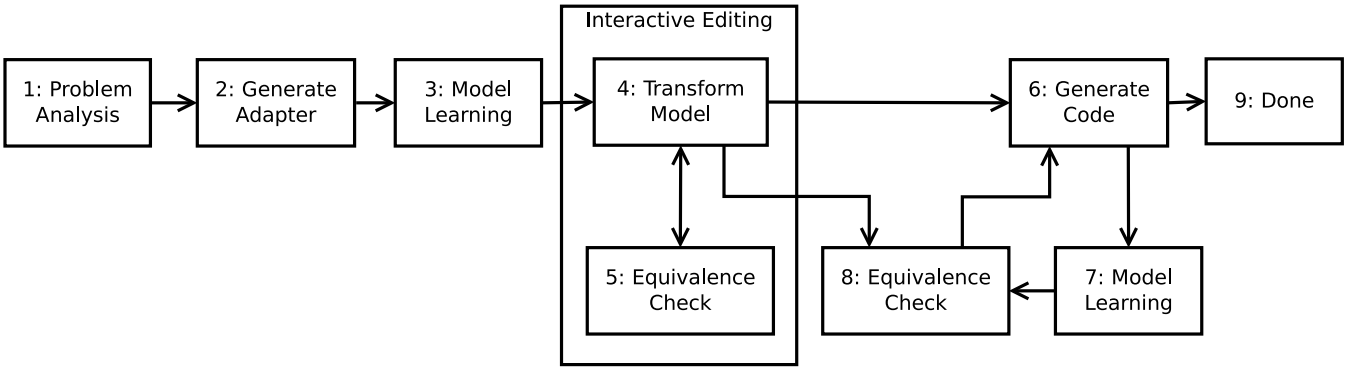

Figure 2. Industrial legacy software rejuvenation toolchain

rejuvenation toolchain. In Figure 2, we have depicted the proposed toolchain. In the subsequent paragraphs, we explain every step of the toolchain.

Step 1 The workflow starts with an analysis of the implementation that is a candidate for refactoring. We determine the external interfaces of the unit that we want to refactor. We use the term unit for a file with source code.

Step 2 To learn a model, the unit needs to be connected to a model learner via an adapter. The adapter connects the external interfaces of the System Under Learning (SUL) to the learner. The adapter makes it also possible to connect the learner to a SUL which is written in another programming language. The adapter can be automatically generated based on the external interfaces of the unit. We use a metaprogramming tool for the adapter generation.

Step 3 A model from the old software is acquired using model learning.

Step 4 Code smells can be removed in this step, e.g. code duplication. The learned model is used as a starting point. This model can be large and might lack an useful structure. Using model to model transformations, new decompositions for the unit can be generated. The user can choose one of the proposed decompositions and start manual editing when required. For instance, the learned state machine will not contain meaningful names for states. This is something that can be added manually.

Step 5 To gain confidence that the transformed model has the same behavior as the learned model, an equivalence checker is used. When the equivalence checker produces a counter example, we go back to *Step 4*.

Step 6 The code smells are removed in this step. From the model, a new implementation is generated. The generated software will be a formfit replacement of the original software, since the external interfaces are the same. When constructing a code generator, we can prevent that code smells are in the generated code.

Step 7 From the newly generated software, a model is acquired using model learning. Since the external interfaces remain unchanged, the adapter we generated in *Step 2* can be reused for learning a model of the newly generated software.

Step 8 For confidence that the generated implementation has the same behavior as the old implementation, an equivalence checker is used. In this step, we can potentially find issues in the code generator of *Step 6*. When the equivalence checker produces a counter example, we need to inspect the code generator.

Step 9 When the generated code is equivalent to the behavior of the old implementation, the generated code can be delivered to the archive and we are done. A new unit can be selected for refactoring and for this unit we start at *Step 1*.

5 First Prototype

We present an artificial case for which we execute all steps of Figure 2 to illustrate our approach. The case is about a vending machine for which we provide a C++ implementation with a few code smells. We choose C++ because it is one of the popular programming languages practiced in the CPS domain[7]. The aim is to generate a formfit replacement of the legacy implementation in which the code smells are removed. Note that our aim for formfit replacement (see also *Req. 4*) implies that the removal of any bugs is out of scope. For every step, we describe which artefacts flow in (its input) and out (its output).

5.1 Step 1: Problem Analysis

Input. The implementation of a vending machine, i.e. the C++ interfaces and C++ implementation.

Listing 1 shows the provided interface. A class VendingMachine is declared together with some public functions. The implementation of the vending machine also relies on some private declarations. It has:

- an enum class for Drink selection;
- a handleOneCoin, handleTwoCoins and handleThreeCoins functions;
- an mCoinsEntered integer variable;

[7] https://www.tiobe.com/tiobe-index/

```
#include "Hardware.h"
class VendingMachine {
public:
  VendingMachine();
  ~VendingMachine() = default;
  void startUp();
  void enterCoin();
  void returnCoins();
  void selectSugar();
  void selectCoffee();
  void makeDrink();
private:
  enum class Drink { NothingSelected, Coffee, Sugar,
CoffeeWithSugar };
  void handleOneCoin();
  void handleTwoCoins();
  void handleThreeCoins();
  int mCoinsEntered;
  Drink mDrink;
  Hardware mHardware;
};
```

Listing 1. Provided Interface

```
class Hardware {
public:
    void started();
    void startFirst();
    void diplayEnterCoin();
    void displayMakeYourChoice();
    void returnCoins();
    void diplayCoffeeSelected();
    void diplaySugarSelected();
    void diplayCoffeeWithSugarSelected();
    void returnCoffee();
    void returnCoffeeWithSugar();
};
```

Listing 2. Required Interface

- an mDrink variable storing the chosen drink.
- an instance of an object that implements the required Hardware interface.

The vending machine has a required interface to drive its hardware. Listing 2 shows this required interface. A class Hardware is declared together with some public functions.

The source code in C++ of the vending machine can be viewed in Appendix A, Listing 11.

The vending machine has the following behavior. It first needs to be started by calling the startUp function on its provided interface. When the vending machine is started, a global variable is set to true. After startup, the vending machine can return coffee with or without sugar, but only when two coins have been inserted. For this, the user first needs to call the enterCoin function twice. The number of coins entered is stored in an mCoinsEntered member variable. After that, the user can call the selectSugar and selectCoffee functions. This choice is stored in the mDrink member variable. Next the user can call the makeDrink function. This function resets the mCoinsEntered and mDrink member variables. For the user, it is possible to get the entered coins back.

The aim is to improve the presented implementation because its maintainability is hampered by the presence of the following code smells described in [23].

- **Code duplication.** The handleOneCoin and handleTwoCoins functions have the same body.
- **Dead code.** The private handleThreeCoins function is never called.
- **Repeated switches.** The selectSugar, selectCoffee and makeDrink functions implement switch-statements. Would we add a new drink such as tea then we need to update these three functions.
- **Magic number.** The code has many if-statements checking if mCoinsEntered is 2. This 2 is a magic number. Ideally, 2 is defined once in a variable declared as a constant. If 2 needs to be changed to for example 4, then we need to change it at 1 place instead of 5 places with the risk of missing a place.
- **Spaghetti code.** The state is coded using the isStarted global variable in combination with the mCoinsEntered and mDrink member variables. This makes the code hard to comprehend.

5.2 Step 2: Generate Adapter

In this step, we create an adapter to connect the System Under Learning (SUL) to the model learner. Most learning algorithms are implemented in LearnLib [19]. However, the latest L# (2022) [33] algorithm is not implemented in LearnLib yet. Since the L# algorithm is implemented in the lsharp_app,[8] we will use this application. This learner is implemented in the Rust programming language while our SUL is in C++. Hence, we need to connect a learner implemented in Rust with a SUL implemented in C++.

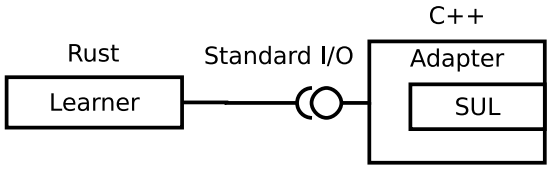

Figure 3. Model-learning setup

As Schuts et. al. have observed in [30], the time the model learner has to wait after supplying an input for the output and to reset the implementation has a large impact on the time the learner needs to learn an implementation. In their

[8] https://gitlab.science.ru.nl/bharat/lsharp_app

```
#include <string>
#include <iostream>
#include "VendingMachine.h"
#include "Hardware.h"
std::string mReturn = "-";
void Hardware::started()
{ mReturn = "Hardware::started()"; }
...
void Hardware::returnCoffeeWithSugar()
{ mReturn = "Hardware::returnCoffeeWithSugar()"; }
void main(void) {
  VendingMachine* sul = new VendingMachine();
  while (true) {
    std::string symb = "";
    getline(std::cin >> std::ws, symb);
    if (symb.length() == 0)  {
      getline(std::cin >> std::ws, symb);
    }
    if (std::string(symb) == "RESET") {
      delete sul;
      sul = new VendingMachine();
    } else {
      mReturn = "-";
      if (std::string(symb) ==
          "VendingMachine::startUp()") {
        sul->startUp();
...
      } else if (std::string(symb) ==
          "VendingMachine::makeDrink()") {
        sul->makeDrink();
      } else {
        std::cout << "\tUnknown symb " << symb << std::endl;
      }
      std::cout << mReturn << std::endl;
    }
  }
}
```

Listing 3. Adapter

```
cargo run -- -I VendingMachine::startUp()
  -I VendingMachine::enterCoin()
  -I VendingMachine::returnCoins()
  -I VendingMachine::selectSugar()
  -I VendingMachine::selectCoffee()
  -I VendingMachine::makeDrink() -M SUL.exe
```

Listing 4. Run batch file

setup, a SUL is placed in a separate executable and the learner connects to this executable using a TCP/IP socket connection. This is sub optimal because it takes a lot of time to send and receive messages over a TCP/IP stack, and to stop and start an executable for a reset.

To improve the learning time, we propose a learning setup as depicted in Figure 3. Both the learner and the adapter that includes the SUL are separate executables. These executables connect to each other using the standard input and output. On a special RESET command, the adapter destructs the SUL and constructs a new instance of the SUL. Using the standard input and output is much faster than sending commands over a TCP/IP connection. Additionally, it is faster to delete and create a new object than to stop and start an executable, and establish a new TCP/IP connection.

Input. The following artefacts are input for the adapter generation:

- Provided interface in Listing 1.
- Required interface in Listing 2.

We use a Rascal script to automatically generate the adapter. Because we need to parse the provided and required C++ interface files, we use Rascal's C++ front-end called ClaiR [1]. The Rascal code can be found in Listing 12 of Appendix B. Listings 3 & 4 provide the output of the Rascal script for the vending machine case.

Listing 3 depicts the adapter. The adapter includes the provided and required interfaces. Next it declares an mReturn string member variable. Then for all public functions in the required interface, a function is declared which stores its name in the mReturn string upon invocation. Because of space limitations, we only printed the first and last function. The main function creates an instance VendingMachine and stores a pointer to the object in the sul variable. When a RESET string is read from the standard input, the VendingMachine object is deleted and a new one created. Other strings are matched with a string describing a function from the provided interface. When there is a match, the function is called on the SUL. After the function call has returned, the contents of the mReturn string is printed to the standard output.

The implementation, adapter, provided interface and required interface are placed in a Microsoft Visual Studio[9] project. The project is compiled into an executable: SUL.exe.

The learner needs to be instructed with an input alphabet. Listings 4 shows the generated batch file we used to utilize the model learner. Listing 12 of Appendix B shows how this batch file is generated.

Output. The following artefacts are the outputs of this step:

- Adapter in Listing 3.
- SUL.exe compilation of Listings 11 & 3.
- Run.bat in Listing 4.

5.3 Step 3: Model Learning

In *Step 3*, we are going to learn a model from the SUL.

Input. Inputs for this step are the outputs of *Step 2*. When we run the Run.bat file on the vending machine case, the learner needs 77 resets and 364 inputs during the MQ phase

[9] https://visualstudio.microsoft.com/

```
// source file : VendingMachine.cpp
// provided interface : VendingMachine.h
// required interface : Hardware.h
digraph g {
  s99 [shape = "circle" label="s99"];
  s76 [shape = "circle" label="s76"];
  s212 [shape = "circle" label="s212"];
  s1 [shape = "circle" label="s1"];
  s0 [shape = "circle" label="s0"];
  s48 [shape = "circle" label="s48"];
  s97 [shape = "circle" label="s97"];
  s0 -> s1 [label="VendingMachine::startUp() /
                Hardware::started()"];
...
  s1 -> s48 [label="VendingMachine::enterCoin() /
                Hardware::diplayEnterCoin()"];
...
  s48 -> s76 [label="VendingMachine::enterCoin() /
                Hardware::diplayEnterCoin()"];
  s48 -> s1 [label="VendingMachine::returnCoins() /
                Hardware::returnCoins()"];
...
  s76 -> s1 [label="VendingMachine::returnCoins() /
                Hardware::returnCoins()"];
  s76 -> s97 [label="VendingMachine::selectSugar() /
                Hardware::diplaySugarSelected()"];
  s76 -> s99 [label="VendingMachine::selectCoffee() /
                Hardware::diplayCoffeeSelected()"];
...
  s97 -> s1 [label="VendingMachine::returnCoins() /
                Hardware::returnCoins()"];
...
  s97 -> s212 [label="VendingMachine::selectCoffee() /
                Hardware::diplayCoffeeWithSugarSelected()"];
...
  s99 -> s1 [label="VendingMachine::returnCoins() /
                Hardware::returnCoins()"];
  s99 -> s212 [label="VendingMachine::selectSugar() /
                Hardware::diplayCoffeeWithSugarSelected()"];
...
  s99 -> s1 [label="VendingMachine::makeDrink() /
                Hardware::returnCoffee()"];
...
  s212 -> s1 [label="VendingMachine::returnCoins() /
                Hardware::returnCoins()"];
...
  s212 -> s1 [label="VendingMachine::makeDrink() /
                Hardware::returnCoffeeWithSugar()"];
__start0 [label="" shape="none" width="0" height="0"];
    __start0 -> s0;
}
```

Listing 5. Learned Model (Removed Self-Transitions)

```
// source file : VendingMachine.cpp
// provided interface : VendingMachine.h
// required interface : Hardware.h
digraph g {
  coffeeSelected [shape = "circle" label="coffeeSelected"];
  sugarSelected [shape = "circle" label="sugarSelected"];
  coffeeWithSugarSelected [shape = "circle"
    label="coffeeWithSugarSelected"];
  started [shape = "circle" label="started"];
  idle [shape = "circle" label="idle"];
  oneCoinEntered [shape = "circle" label="oneCoinEntered"];
  twoCoinsEntered
    [shape = "circle" label="twoCoinsEntered"];
  idle -> started [label="VendingMachine::startUp() /
                Hardware::started()"];
...
  coffeeWithSugarSelected -> started
                [label="VendingMachine::makeDrink() /
                Hardware::returnCoffeeWithSugar()"];
__start0 [label="" shape="none" width="0" height="0"];
    __start0 -> idle;
}
```

Listing 6. Transformed Model

and 2530255 resets and 41164449 inputs during the EQ phase. This results in a state machine with 7 states and 42 transitions. The output of the model learner is the Dot file [14], partly shown in Listing 5. Because of its size, we removed the self-transitions. Observe that dead code (function handleThreeCoins) is not present in the learned model.

Output. The following artefact is output of this step:

- Learned Dot file in Listing 5.

5.4 Step 4: Transform Model

In this step, we edit the learned state machine. For the moment, we just change the Dot file manually; more advanced editing support is part of future work.

Input. The input for this is the learned Dot file.

For our case, we apply a trivial editing by only renaming the state names. Listing 6 depicts part of the new model, showing only the first and last transition.

Output. The following artefact go the the next step:

- Edited Dot file in Listing 6.

5.5 Step 5: Equivalence Checking

The equivalence of the learned and edited models can be determined using a model checker. For our case, we use the mCRL2 model checker [16].

Input. The two Dot files from *Step 4* are the inputs for this step.

To perform an equivalence check using mCRL2, we need two models in the mCRL2 language. These two models are

```
sort
States = struct Sidle | Sstarted | SoneCoinEntered |
  StwoCoinsEntered | SsugarSelected |
  ScoffeeSelected | ScoffeeWithSugarSelected;
Events = struct EstartUp | EenterCoin | EreturnCoins |
  EselectSugar | EselectCoffee | EmakeDrink;
Actions = struct Astarted | AstartFirst |
  AdiplayEnterCoin | AreturnCoins |
  AdisplayMakeYourChoice | AdiplaySugarSelected |
  AdiplayCoffeeSelected | AdiplayCoffeeWithSugarSelected |
  AreturnCoffee | AreturnCoffeeWithSugar;
act iProvided:Events;
act iRequired:Actions;
proc Spec(s:States) =
  (s==SoneCoinEntered) -> (
    iProvided(EstartUp) . iRequired(Astarted) .
      Spec(SoneCoinEntered) +
    iProvided(EenterCoin) . iRequired(AdiplayEnterCoin) .
      Spec(StwoCoinsEntered) +
    iProvided(EreturnCoins) . iRequired(AreturnCoins) .
      Spec(Sstarted) +
    iProvided(EselectSugar) . iRequired(AdiplayEnterCoin) .
      Spec(SoneCoinEntered) +
    iProvided(EselectCoffee) . iRequired(AdiplayEnterCoin) .
      Spec(SoneCoinEntered) +
    iProvided(EmakeDrink) . iRequired(AdiplayEnterCoin) .
      Spec(SoneCoinEntered)) +
...
  (s==StwoCoinsEntered) -> (
    iProvided(EstartUp) . iRequired(Astarted) .
      Spec(StwoCoinsEntered) +
    iProvided(EenterCoin) .
    iRequired(AdisplayMakeYourChoice) .
      Spec(StwoCoinsEntered) +
    iProvided(EreturnCoins) . iRequired(AreturnCoins) .
      Spec(Sstarted) +
    iProvided(EselectSugar) .
    iRequired(AdiplaySugarSelected) .
      Spec(SsugarSelected) +
    iProvided(EselectCoffee) .
    iRequired(AdiplayCoffeeSelected) .
      Spec(ScoffeeSelected) +
    iProvided(EmakeDrink) .
    iRequired(AdisplayMakeYourChoice) .
      Spec(StwoCoinsEntered));
init Spec(Sidle);
```

Listing 7. mCRL2 Model

generated from the learned and edited Dot files using a Rascal script. Listing 7 shows parts of the resulting mCRL2 model when translating the Dot file in Listing 6 to mCRL2. The model starts by defining structs for the states, events and actions. Next the events are related to the iProvided interface and actions to the iRequired interface. Then the Spec process is defined. Per state, all events are listed including

```
mcrl22lps.exe edited.mcrl2 edited.lps --verbose
lps2lts.exe edited.lps edited.lts --verbose
mcrl22lps.exe final.mcrl2 final.lps --verbose
lps2lts.exe final.lps final.lts --verbose
ltscompare.exe edited.lts final.lts --counter-example
  --equivalence=branching-bisim --verbose
```

Listing 8. mCRL2 Commands

the resulting action of an event and the next state. On the last line, the Spec is initialized with the Sidle state.

In Appendix C, we provide the Rascal scripts to translate the Dot files to mCRL2 models. In Listings 13 & 14, we describe the concrete syntax of our Dot language dialect as described in Section 5.3. The Rascal code that performs the translation can be found in Listings 15 & 16.

To check equivalence, we execute the commands provided in Listing 8. We first convert the mCRL2 files to Linear Process Specifications (LPS) files from which we can generate Labeled Transition System (LTS) files. With the LTS files, we can compare the two models using strong bisimulation as introduced in Section 3. The outcome of the equivalence check will be a pass or a fail. When it fails, the tool can provide a counter example. This counter example is a sequence of events and actions that leads to a difference in result. A counter example can be presented, for instance, as a UML sequence diagram to the user of the toolchain [27].

In our example case, we compared Listing 5 and Listing 6 and the equivalence check passed, since the externally visible behavior of the two state machines is the same.

Output.

- Pass or fail. In case of a fail, a counter example is provided.

5.6 Step 6: Generate Code

When the model is edited (*Step 4*) and the equivalence check holds (*Step 5*), we generate a new implementation.

Input. The following artefact is input for this step:

- Edited Dot file in Listing 6.

For our new implementation, we use Rascal to generate a new provided interface as shown in Listing 9. In this interface, the public functions are the same as in the original interface. However, because the implementation is different, the private section is different as well. Concerning the code generator for the new implementation, there are many ways to represent the state machine[10]. We have chosen to create the private functions createTransition and triggerTransition. As shown in Listing 9, the createTransition function takes a State enumeration as first and fourth input arguments. It also takes an Event enumeration as its second argument. The

[10]We were inspired by the examples from https://stackoverflow.com/questions/1647631/c-state-machine-design/1647679#1647679

```
#include "Hardware.h"
class VendingMachine {
public:
  VendingMachine();
  ~VendingMachine() = default;
  void startUp();
  void enterCoin();
  void returnCoins();
  void selectSugar();
  void selectCoffee();
  void makeDrink();
private:
  enum class State { Sidle, Sstarted, SoneCoinEntered,
    StwoCoinsEntered, SsugarSelected, ScoffeeSelected,
    ScoffeeWithSugarSelected, nrOfValues };
  enum class Event { EstartUp, EenterCoin, EreturnCoins,
    EselectSugar, EselectCoffee, EmakeDrink, nrOfValues };
  typedef void(Hardware::*Action)(void);
  void createTransition(State currentState, Event event,
    Action action, State nextState);
  void triggerTransition(Event event);
  State mCurrentState;
  std::vector< std::tuple<Action, State> > mTransitions;
  Hardware mHardware;
};
```

Listing 9. Generated C++ Interface

enumerations are declared just above the declaration of the createTransition function. The third argument is an Action. An Action is a function pointer pointing to a public function of the required Hardware interface. There are three member variables: one for storing the current state, one for storing the transitions, and another one for a reference to the object that implements the required interface.

In Listing 10, we provide part of the new implementation. In the constructor (lines 3–29) all transitions are created (in the listing we show it for the first two states) and it initializes the mCurrentState member variable to the Sidle state (line 3). The implementation of the private createTransition function (lines 42–47) is responsible for creating and storing a transition. Based on the current state and event, it calculates a position in the mTransitions vector (kind of array) and inserts a tuple with the action and next state at the calculated position. On lines 30–41, the provided functions are implemented by calling the triggerTransition function. The triggerTransition function (lines 48–53) calculates the position of the transition in the mTransitions vector based on the current state and the event. It stores the transition tuple in the transition variable. Next the function pointer associated with the transition is invoked. Finally, the mCurrentState member variable is updated (line 52).

Listings 17 & 18 in Appendix D presents the Rascal code for generating a new implementation and the accompanying

```
#include "VendingMachine.h"
#include "Hardware.h"
VendingMachine::VendingMachine() :
mCurrentState(State::Sidle), mHardware() {
    createTransition(State::Sidle, Event::EstartUp,
      &Hardware::started, State::Sstarted);
    createTransition(State::Sidle, Event::EenterCoin,
      &Hardware::startFirst, State::Sidle);
    createTransition(State::Sidle, Event::EreturnCoins,
      &Hardware::startFirst, State::Sidle);
    createTransition(State::Sidle, Event::EselectSugar,
      &Hardware::startFirst, State::Sidle);
    createTransition(State::Sidle, Event::EselectCoffee,
      &Hardware::startFirst, State::Sidle);
    createTransition(State::Sidle, Event::EmakeDrink,
      &Hardware::startFirst, State::Sidle);
    createTransition(State::Sstarted, Event::EstartUp,
      &Hardware::started, State::Sstarted);
    createTransition(State::Sstarted, Event::EenterCoin,
      &Hardware::diplayEnterCoin, State::SoneCoinEntered);
    createTransition(State::Sstarted, Event::EreturnCoins,
      &Hardware::diplayEnterCoin, State::Sstarted);
    createTransition(State::Sstarted, Event::EselectSugar,
      &Hardware::diplayEnterCoin, State::Sstarted);
    createTransition(State::Sstarted, Event::EselectCoffee,
      &Hardware::diplayEnterCoin, State::Sstarted);
    createTransition(State::Sstarted, Event::EmakeDrink,
      &Hardware::diplayEnterCoin, State::Sstarted);
...
}
void VendingMachine::startUp()
{ triggerTransition(Event::EstartUp); }
void VendingMachine::enterCoin()
{ triggerTransition(Event::EenterCoin); }
void VendingMachine::returnCoins()
{ triggerTransition(Event::EreturnCoins); }
void VendingMachine::selectSugar()
{ triggerTransition(Event::EselectSugar); }
void VendingMachine::selectCoffee()
{ triggerTransition(Event::EselectCoffee); }
void VendingMachine::makeDrink()
{ triggerTransition(Event::EmakeDrink); }
void VendingMachine::createTransition(State currentState,
    Event event, Action action, State nextState) {
  auto index = mTransitions.begin() +
    ((int)currentState * (int)Event::nrOfValues)
      + (int)event;
  mTransitions.insert(index,
    std::make_tuple(action, nextState));
}
void VendingMachine::triggerTransition(Event event) {
  auto transition = mTransitions[((int)mCurrentState *
    (int)Event::nrOfValues) + (int)event];
  (mHardware.*std::get<0>(transition))();
  mCurrentState = std::get<1>(transition);
}
```

Listing 10. Generated C++ Implementation

interface header file. It takes the edited Dot file and uses the Dot syntax of Listings 13 & 14 to parse this file.

Output. The following artefacts go to the next step:

- New provided interface in Listing 9.
- New implementation in Listing 10.

5.7 Step 7: Model Learning

Input. The outputs of the previous step are the inputs for this step. In addition, we use the required Hardware interface and reuse the adapter of *Step 2*.

We utilize model learning to check that the generated file has equivalent behavior as the edited Dot file and is thus also equivalent with the original learned model from *Step 3*. The new implementation, new provided interface, adapter and required interface are placed in a Microsoft Visual Studio project and a SUL.exe executable is compiled.

Using SUL.exe, the model learner is executed as described in Section 5.3. The model learning results in a new Dot file. The learned state machine has 7 states and 42 transitions. These values are the same as for the learned state machine from *Step 3*. We do not show this file as it looks very similar to the one in Listing 5.

Output. The following artefact is the output of this step:

- Newly learned Dot file.

5.8 Step 8: Equivalence Checking

Input. The following artefacts go to this step:

- Edited Dot file in Listing 6.
- Newly learned Dot file of *Step 7*.

Here we follow exactly the same steps as described in Section 5.5. Assuming a correct code generator, we expect the equivalence check to pass.

Output.

- Pass or fail. In case of a fail, a counter example is provided which is useful to debug the code generator.

5.9 Step 9: Done

Input. The following artefacts are inputs to this step:

- New provided interface in Listing 9.
- New implementation in Listing 10.

In this step, the generated provided interface and implementation need to be integrated in the target code base. When done with this step, one can start rejuvenating another unit at the start of our toolchain in *Step 1*.

Per code smell that was present in the original implementation, we describe how it is removed in the rejuvenated version of the source code in Listing 10.

- **Code duplication.** Since learning algorithms merge equivalent states, the code duplication in the original code is not present in the new implementation.
- **Dead code.** Dead code is not touched by the learner and therefore did not end up in the learned model and the resulting implementation.
- **Repeated switches.** In our code generator, we do not generate code that make use of switch-statements.
- **Magic number.** The code generator does generate any number, so also not magic numbers.
- **Spaghetti code.** The current state is stored in one mCurrentState member variable.

6 Discussion

In this section, we identify research challenges, we discuss our approach and describe considerations for the steps in our approach.

6.1 Step 1: Problem Analysis

For this paper we kept our case small, but one can imagine that the implementation one wants to rejuvenate has the **large unit** code smell (i.e. when a file contains more than 1000 lines) and **long method** code smell (i.e. when a method has more than 100 lines). In Section 6.6, we describe how a code generator can also remove these code smells.

We choose C++ for our case, but we could as well have chosen C or C# which are also popular in the CPS domain. The approach can also be applied to other programming languages, as long as there is a clear distinction between an implementation, and its provided and required interfaces.

6.2 Step 2: Generate Adapter

As a metaprogramming tool, we choose to use Rascal to parse files containing the provided and required interfaces. Because our SUL is in C++, we used ClaiR, Rascal's C/C++ front-end. We can also use ClaiR for a SUL implemented in the C programming language. However, when the SUL is in another programming language, we should choose another front-end for Rascal or choose another way of generating the adapter. For Rascal there are also language front-ends available for Java and Ada, other languages used in the CPS domain.

The SUL described in Section 5 always invokes exactly one function on the required interface when stimulating the vending machine with a function call on its provided interface. This is a simplification; in general, we have to deal with any number of function calls on the required interface(s) after a single stimulus on the provided interface. Observe that the adapter would return a "-" when no function is called on the required interface of the SUL when a function is called in its provided interface. To generalize our adapter for multiple function calls, we could return a string with a list of function calls. To support zero or multiple function calls on the required interface, we also need change the transition strings of the Dot file which has consequences for the interfaces to the *Steps 5 & 8* and *Steps 6*. Moreover in

Steps 6, the code generator needs to be adapted to handle the zero or multiple function calls on the required interface.

The assumption is that the adapter stubs all required interfaces, including for instance system calls to the operating system. Hence, all function calls by the implementation to be learned will result in an output string. From this assumption, it follows that the compiled SUL.exe will not have side-effects when it is executed by the learner.

In our case, we used functions on the provided and required interfaces with a void return and no parameters in the argument list. Future work for this step is to create a generic adapter generator that can handle functions with return types and parameters, and to do this for multiple programming languages.

6.3 Steps 3 & 7: Model Learning

Since there are many manipulation and visualization tools that support Dot files this could be the format of choice for our tool. Regarding *Steps 3 & 7* in the toolchain, it should be easy to replace the lsharp_app by another model learning tool that implements different learning algorithms such as LearnLib. Both L# and LearnLib produce Dot files as output.

The first three lines of the Dot file (in Listing 5) are not generated by the model learner, but are added manually because this information is required in subsequent steps. We have chosen to add them as comments according the language definition of GraphViz[11]. The benefit of this approach is that standard dot visualization tools still accept this file.

Duhaiby et al. observed in [4] that it could take substantial time to learn a model from an implementation. For this reason, we need to scope the problem domain for our approach and improve the learning setup.

We think our first version can learn models from industrial software with the following limitations:

- **Single Threaded** The unit shall be single threaded. The reason for this is that if there are more threads the number of possible states could potentially explode –because of all possible interleavings between threads– which makes it impossible to learn.
- **Data Independent** Data provided to the unit may not lead to control decisions. We have this restriction because if we need to take all values of a possible variable into account this would lead to at least one state per value of a variable. This again would potentially explode the number of states.

Model learning algorithms need to improve to overcome these limitations. With improved learning algorithms, the single threaded and data independent control units constraints might change. For instance, we could climb up the hierarchy of Figure 4 or include data. We have observed the development of new algorithms such as the TTT (2014) [18] and L# (2022) [33]. This gives us the indication that in the future with improved algorithms, we can learn larger subassemblies of programs.

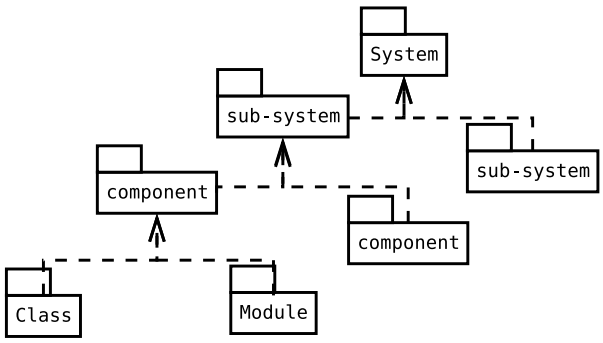

Figure 4. Usage levels

Despite the described limitations, model learning has been applied in industry to learn parts of CPS[12]. At AMSL, a company that produces high-end lithography systems, 33 models have been learned. Ten models have been learned from medical systems at Philips. At high-end printer copier company Canon, 1 very large model has been learned [24]. Hence, *Steps 1–3* of our toolchain have already been performed in industry.

Another approach would be to apply the toolchain many times instead of learning ever larger portions of the source code. This would not change the design of the rejuvenated implementation, but only the contents of the units of which it is composed. Learning smaller units also has the benefit that the learned models can more easily checked by the user of the tool for their correctness.

In Section 5 *Step 2*, we describe a novel way of creating a learning setup that is more efficient than the common approach [30]. The time to learn an implementation depends on the time to perform MQs and EQs, and the time required to reset an implementation. Furthermore, this time depends on the number of states and the number of inputs. In our setup, we decrease the query and reset time because the SUL is no longer in separate executable and connected via standard I/O to the learner instead of TCP/IP. Since model learning cannot learn models of computations or algorithms on data [5], our toolchain cannot rejuvenate all parts of a CPS.

6.4 Step 4: Transform Model

In Section 5.4, we only edited the state names for the case. For more complex programs, the learned model can become quite large. To manage such models, it is useful to decompose them. For this, we want to use algorithms to propose different decompositions to the user of the toolchain. The tool does not have domain knowledge, so choosing the right

[11] https://www.graphviz.org/doc/info/lang.html

[12] https://automata.cs.ru.nl/

decomposition is the responsibility of the user. We see the chosen decomposition as a starting point for manual editing.

We think graph-based algorithms can be applied to propose new decompositions to handle complexity and remove the code smells.

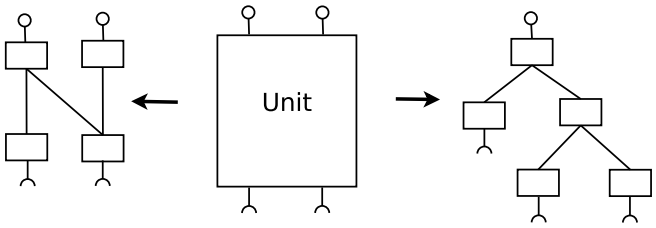

Figure 5. Two ways of decomposing the learned model

Figure 5 shows two ways to split up an unit. The unit in the middle of the figure is split up using either:

- Krohn & Rhodes algorithm [22], see the left drawing. This algorithm creates a cascading decomposition of the unit where the output of one automaton is fed as input to another automaton.
- Biggar et al. algorithm [8], see the right drawing. Other than the Krohn & Rhodes algorithm, this algorithm creates an hierarchy of new units. The algorithm is based on graph theory about modular decomposition. In earlier work, modular decomposition has been applied to digraphs. For instance, in their paper, Biggar et al. describe that digraphs are very similar to finite state machines and they describe the translation.

As mentioned, the input and output of this step are Dot files. As long as we can translate the format of an algorithm or tool in this step to and from a Dot file, we can integrate such an algorithm or tool in our toolchain.

For our first version of the toolchain, we have chosen for the Dot file format. In future work, we will investigate whether this format is expressive enough. The Dot file format does support a notation to describe hierarchical state machines, but is it also possible to express the decompositions as generated by the Krohn & Rhodes algorithm?

Another challenge for this step is to provide a user friendly graphical user interface to edit and change the models.

6.5 Step 5: Equivalence Checking

In *Step 5*, we employ equivalence checking to make sure that changing the learned model does not change the externally visible behavior of the implementation.

When the number of states are in the millions, equivalence checkers can suffer from state space explosion. But we do not expect to encounter state space explosion with our approach, because today learning an unit of millions of states is impossible given the time required to query and reset the SUL.

From Dot files, it is very easy to write model generators. We created one for mCRL2, our equivalence checker. With little effort we could create a model for another equivalence checker. With this approach, we can easily change our tool to use another equivalence checker without affecting the other steps in our toolchain.

6.6 Step 6: Generate Code

In the code generator described in Section 5.6, the constructor could become quite large if many transitions need to be created. The number of generated transitions is the number of states times the number of events. Hence, with 10 states and 10 transitions, we would violate the **long method** code smell (i.e. when a method has more than 100 lines). To prevent this, we could add a check in the code generator that if the number of createTransition function calls becomes larger than 100 that we create a private function per state and call these private functions from the constructor. For this to become too long, we need to learn a system with more than 100 events and the number of states should be less than 100. Another way to deal with the long constructor is to generate a completely different implementation based on the object oriented state pattern as described by Gamma et.al. [13].

Another code smell is **large unit** i.e. when a file contains more than 1000 lines. Our current generator leads to two lines per event plus eleven lines for the private functions (lines 42–53 of Listing 10). When the constructor is shorter than 100 lines, the complete unit will not be larger than 1000 lines. Should the file contain more than 1000 lines, then the file can be split up into one file per state as done in the object oriented state pattern. Hence, the code generator can be adapted such that the generated code is free of the *large unit* code smell.

In Section 5.9, we described that in the rejuvenated version of the source code in Listing 10 the **code duplication**, **dead code**, **repeated switches**, **magic number** and **spaghetti code** code smells are removed. Further research need to take place on more cases to check if we can generalize our results and if there are other code smells that can potentially be removed by the toolchain.

The *Steps 2 & 6* are the only steps that are programming language specific. Hence, if we want to support another programming language, we need to change the code generators or tools in these steps.

As mentioned in Section 6.2, our current version only supports functions with an empty parameter list and a void return. We could extend the support with functions that pass data from the provided interface to the required interface.

An alternative is the use of a state-based tool such as Rhapsody, Visual State, ASD, Cocotec or Dezyne which are described in Section 3. We could, for instance, choose Dezyne because of its open source release. Instead of generating code, we need to generate a Dezyne model from which the Dezyne

tool can generate source code in multiple programming languages. The advantage is that we only need to create one code generator towards the Dezyne language to be able to generate code for multiple programming languages. An additional benefit is that we can maintain the Dezyne model and have a more abstract way to describe the behavior of the system instead of maintaining its generated code.

7 Concluding Remarks

In this paper, we presented our vision of a toolchain for the semi-automated code smell removal in industrial stateful legacy software. In Section 1, we identified four requirements for the toolchain and here we describe how they are met.

We target our tool to refactor industrial software of Cyber Physical Systems (CPS) and generate state machines which are commonly used in these kind of stateful systems. The toolchain combines existing techniques such as model learning to acquire a state machine of the behavior of a software unit, graph based algorithms to interactively propose restructuring of the learned state machine, and equivalence checking to test if the changes of the state machine are in fact a refactoring. With this we satisfy *Req. 1* for state based CPS and *Req. 2* for supporting multiple programming languages by means of model learning.

The toolchain can remove many code smells as required by *Req. 3*. By learning the behavior of the legacy code, smells like dead code and code duplication are removed automatically. Other code smells can be removed or avoided by the code generator.

Compared to existing refactoring tools, our proposed toolchain provides confidence –by means of equivalence checking– that the code after refactoring behaves the same as before refactoring. Because of this, the toolchain satisfies *Req. 4* as well.

We described a case for which we executed all steps of a preliminary version of the toolchain. In addition, we described challenges for further generalization of the toolchain. One of these challenges is to increase the capabilities of model learning algorithms to deal with multi-threaded and data-dependent control units. To remove these restrictions, model learning algorithms need to be improved. Since the learned models can be quite big and complex, new algorithms need to be developed to restructure these learned state machines. Furthermore, we expect challenges in making the toolchain modular. We described a first attempt to define interfaces between the steps of the toolchain in such a way that we can replace a tool or algorithm by another one without impacting the previous or next steps.

As future work, we want to make the step towards industry. Potts describes an approach which he calls "industry-as-lab" [26]. With this approach the researcher and the industrial partner work closely together where the research results are iteratively tried and implemented. While applying this approach, we want to gain feedback and improve our toolchain.

Acknowledgments

We would like to thank Bharat Garhewal for the creation and support on the lsharp_app.

The anonymous reviewers are gratefully acknowledged for their many useful comments and suggestions.

A Source Code of The Vending Machine

```
#include "VendingMachine.h"
#include "Hardware.h"
bool isStarted;
VendingMachine::VendingMachine() :mCoinsEntered(0),
  mDrink(Drink::NothingSelected), mHardware()
{ isStarted = false; }
void VendingMachine::startUp() {
  if (!isStarted) isStarted = true;
  mHardware.started();
}
void VendingMachine::enterCoin() {
  if (mCoinsEntered <= 1) handleOneCoin();
  else handleTwoCoins();
}
void VendingMachine::handleOneCoin() {
  if (!isStarted) { mHardware.startFirst(); return; }
  if (mCoinsEntered == 2) mHardware.displayMakeYourChoice();
  else { mCoinsEntered++; mHardware.diplayEnterCoin(); }
}
void VendingMachine::handleTwoCoins() {
  if (!isStarted) { mHardware.startFirst(); return; }
  if (mCoinsEntered == 2) mHardware.displayMakeYourChoice();
  else { mCoinsEntered++; mHardware.diplayEnterCoin(); }
}
void VendingMachine::handleThreeCoins() {
  if (!isStarted) { mHardware.startFirst(); return; }
  if (mCoinsEntered == 3) mHardware.displayMakeYourChoice();
  else { mCoinsEntered++; mHardware.diplayEnterCoin(); }
}
void VendingMachine::returnCoins() {
  if (!isStarted) { mHardware.startFirst(); return; }
  if (mCoinsEntered > 0) { mCoinsEntered = 0; mDrink =
    Drink::NothingSelected; mHardware.returnCoins();
  } else mHardware.diplayEnterCoin();
}
void VendingMachine::selectSugar() {
  if (!isStarted) { mHardware.startFirst(); return; }
  if (mCoinsEntered == 2)
    switch (mDrink) {
      case Drink::NothingSelected: mDrink = Drink::Sugar;
        mHardware.diplaySugarSelected(); break;
      case Drink::Sugar:
        mHardware.diplaySugarSelected(); break;
      case Drink::Coffee: mDrink = Drink::CoffeeWithSugar;
        mHardware.diplayCoffeeWithSugarSelected(); break;
      case Drink::CoffeeWithSugar:
        mHardware.diplayCoffeeWithSugarSelected(); break;
```

```
  } else mHardware.diplayEnterCoin();
}
void VendingMachine::selectCoffee() {
  if (!isStarted) { mHardware.startFirst(); return; }
  if (mCoinsEntered == 2)
    switch (mDrink) {
      case Drink::NothingSelected: mDrink = Drink::Coffee;
        mHardware.diplayCoffeeSelected(); break;
      case Drink::Sugar: mDrink = Drink::CoffeeWithSugar;
        mHardware.diplayCoffeeWithSugarSelected(); break;
      case Drink::Coffee:
        mHardware.diplayCoffeeSelected(); break;
      case Drink::CoffeeWithSugar:
        mHardware.diplayCoffeeWithSugarSelected(); break;
  } else mHardware.diplayEnterCoin();
}
void VendingMachine::makeDrink() {
  if (!isStarted) { mHardware.startFirst(); return; }
  if (mCoinsEntered == 2)
    switch (mDrink) {
      case Drink::NothingSelected:
        mHardware.displayMakeYourChoice(); break;
      case Drink::Sugar:
        mHardware.displayMakeYourChoice(); break;
      case Drink::Coffee: mCoinsEntered = 0;
        mDrink = Drink::NothingSelected;
        mHardware.returnCoffee(); break;
      case Drink::CoffeeWithSugar: mCoinsEntered = 0;
        mDrink = Drink::NothingSelected;
        mHardware.returnCoffeeWithSugar(); break;
  } else mHardware.diplayEnterCoin();
}
```

Listing 11. Source code

B Adapter Generator

```
module Adapter
import lang::cpp::AST;
import IO;
import String;
public void main() {
  iProvidedInterface = |project://Step2/code/VendingMachine.h|;
  iRequiredInterface = |project://Step2/code/Hardware.h|;
  oAdapterFile = |project://OnwardsStep2/code/new/Adapter.cpp|;
  generate(iProvidedInterface, iRequiredInterface,
           oAdapterFile, oBatFile);
}
void generate(iProvidedInterface, iRequiredInterface,
              oAdapterFile,oBatFile) {
  rVal = "";
  lrel[str, str, str, str] sm = [];
  astProvidedInterface = parseCpp(iProvidedInterface);
  astRequiredInterface = parseCpp(iRequiredInterface);
  rVal =
    "#include \<string\>
    '#include \<iostream\>
    '#include \"<iProvidedInterface.file>\"
    '#include \"<iRequiredInterface.file>\"
    'std::string mReturn = \"-\";
    '<for (n <- getFunctions(astRequiredInterface)) {>
    'void <getClassName(astRequiredInterface)>::<n> {
      mReturn =
        \"<getClassName(astRequiredInterface)>::<n>\"; }<}>
    'void main(void) {
    '  <getClassName(astProvidedInterface)>*
         sul = new <getClassName(astProvidedInterface)>();
    '  while (true) {
    '    std::string symb = \"\";
    '    getline(std::cin \>\> std::ws, symb);
    '    if (symb.length() == 0)     {
    '      getline(std::cin \>\> std::ws, symb);
    '    }
    '    if (std::string(symb) == \"RESET\") {
    '      delete sul;
    '      sul = new <getClassName(astProvidedInterface)>();
    '    } else {
    '      mReturn = \"-\";
    '      <for (n <- getFunctions(astProvidedInterface)) {>
      if (std::string(symb) ==
      \"<getClassName(astProvidedInterface)>::<n>\") {
    '       sul-\><n>;
    '      } else <}>{
    '       std::cout \<\< \"\\tUnknown symb \"
      \<\< symb \<\< std::endl;
    '      }
    '      std::cout \<\< mReturn \<\< std::endl;
    '    }
    '  }
    '}";
  writeFile(oAdapterFile, rVal);
  rVal = "cargo run -- <for (e <-
    getFunctions(astProvidedInterface)) {> -I
      <getClassName(astProvidedInterface)>::<e><}>
        -M SUL.exe";
  writeFile(oBatFile, rVal);
}
str getClassName(ast) {
  return ["<n>" | /class([], name(n), [], [*_])
    := ast ][0];
}
list[str] getFunctions(ast) {
  rVal = [];
  visit(ast) {
    case class([], name(c), [], body): {
      isPublicSection = false;
      for (b <- body) {
        visit (b) {
          case visibilityLabel(\public()):
            isPublicSection = true;
          case visibilityLabel(\private()):
            isPublicSection = false;
          case visibilityLabel(\protected()):
            isPublicSection = false;
          case functionDeclarator([], [], name(n), [], []):
            if (isPublicSection && !contains("<n>", "<c>"))
              rVal += "<n>()";
```

```
        }
      }
    }
  }
  return rVal;
}
```

Listing 12. Adapter Generator in Rascal

C mCRL2 Generator

```
module Syntax
layout Layout = [\ \t\n\r] !>> [\ \t\n\r];
start syntax Build = build: Sources sources
  Digraph digraph;
syntax Sources = sources: SourceFile sourceFile
  ProvidedInterface providedInterface
  RequiredInterface requiredInterface;
syntax SourceFile = sourceFile: "//" "source" "file" ":"
  Id file;
```

Listing 13. Dot Syntax in Rascal

```
syntax ProvidedInterface = providedInterface:
  "//" "provided" "interface" ":" Id file;
syntax RequiredInterface = requiredInterface:
  "//" "required" "interface" ":" Id file;
syntax Digraph = digraph: "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 action
  "\"];";
syntax Footer = footer: "__start0" "[" "label" "=" "\"\""
  "shape" "=" "\"none\"" "width" "=" "\"0\""
  "height" "=" "\"0\"" "];" InitialState initialState;
syntax InitialState = initialState: "__start0" "-\>"
  Id firstState ";";
lexical Id = ([a-zA-Z/.\-][a-zA-Z0-9_/.:()]* !>>
  [a-zA-Z0-9_/.:()]) \ Reserved;
keyword Reserved = "digraph" | "g" | "State" | "," | ";" |
  "=" | "source" | "file" | "provided" | "required" |
  "interface" | "{" | "}" | "[" | "]" | "on" | "state";
```

Listing 14. Dot Syntax in Rascal

```
module Mcrl2Model
import IO;
...
import Syntax;
public void main() {
  iFile = |project://Step5/code/edited.dot|;
  oFile = |project://Step5/code/edited.mcrl2|;
  mCrl2Model(iFile, oFile);
  iFile = |project://Step5/code/learned.dot|;
  oFile = |project://Step5/code/learned.mcrl2|;
  mCrl2Model(iFile, oFile);
}
void mCrl2Model(iFile, oFile) {
  rVal = "";
  lrel[str, str, str, str] sm = [];
  cst = parse(#start[Build], iFile);
  sm = strip(getStateMachine(cst));
  states = dup([ c | <c, _, _, _> <- sm ]);
  events = dup([ e | <_, e, _, _> <- sm ]);
  actions = dup([ a | <_, _, a, _> <- sm ]);
  rVal =
    "sort
    'States = struct S<intercalate(" | S", states)>;
    'Events = struct E<intercalate(" | E", events)>;
    'Actions = struct A<intercalate(" | A", actions)>;
    'act iProvided:Events;
    'act iRequired:Actions;
    'proc Spec(s:States) =
    '    <printSM(sm)>
    'init Spec(S<getInitialState(cst)>);
    '";
  writeFile(oFile, rVal);
}
```

Listing 15. mCRL2 Generator in Rascal

```
str printSM(sm) {
  rVal = "";
  for (currentState <-
     { currentState | <currentState, _, _, _> <- sm } ) {
    rVal += "(s==S<currentState>) -\> (\n";
    trans = [];
    for (<c, e, a, n> <- sm, currentState == c) {
      trans += "   iProvided(E<e>) . iRequired(A<a>) .
        Spec(S<n>)";
    }
    rVal += intercalate(" +\n", trans) + ") +\n";
  }
  return rVal[..-3] + ";";
}
...
```

Listing 16. mCRL2 Generator in Rascal

D Implementation Generator

```
module NewImplementation
import IO;
...
import Syntax;
public void main() {
  rVal = "";
  lrel[str, str, str, str] sm = [];
  iFile = |project://Step6/code/edited.dot|;
  cst = parse(#start[Build], iFile);
  oSourceFile = |project://Step6/code/new| +
    getSourceFileName(cst);
  oProvidedInterface = |project://Step6/code/new| +
    getProvidedInterfaceName(cst);
  sm = getStateMachine(cst);
  states = dup([ c | <c, _, _, _> <- sm ]);
  events = dup([ e | <_, e, _, _> <- sm ]);
  strippedEvents = [ split("::", e)[1] | e <- events ];
```

```
  actions = dup([ a | <_, _, a, _> <- sm ]);
  iProvided = split("::", events[0])[0];
  iRequired = split("::", actions[0])[0];
  rVal =
    "#include \"<getProvidedInterfaceName(cst)>\"
    '#include \"<getRequiredInterfaceName(cst)>\"
    '<iProvided>::<iProvided>() :
      mCurrentState(State::S<getInitialState(cst)>),
      m<iRequired>() {
    '    <createTransitions(sm, iRequired)>}
    '<triggerTransitions(events)>
    'void <iProvided>::createTransition(State currentState,
      Event event, Action action, State nextState) {
    '    auto index = mTransitions.begin() + ((int)
      currentState * (int)Event::nrOfValues) + (int)event;
    '    mTransitions.insert(index, std::make_tuple(action,
      nextState));
    '}
```

Listing 17. Implementation Generator in Rascal

```
    'void <iProvided>::triggerTransition(Event event) {
    '    auto transition = mTransitions[((int)mCurrentState
      * (int)Event::nrOfValues) + (int)event];
    '    (m<iRequired>.*std::get\<0\>(transition))();
    '    mCurrentState = std::get\<1\>(transition);
    '}
    '";
  writeFile(oSourceFile, rVal);
  rVal =
     '#include \"<getRequiredInterfaceName(cst)>\"
     'class <iProvided> {
     'public:
     '    <iProvided>();
     '    ~<iProvided>() = default;
     '    <for (e <- strippedEvents) {>
     '    void <e>();<}>
     'private:
     '    enum class State { S<intercalate(", S", states)>,
       nrOfValues };
     '    enum class Event { E<intercalate(", E",
       strippedEvents)>, nrOfValues };
     '    typedef void(<iRequired>::*Action)(void);
     '    void createTransition(State currenState,
       Event event, Action action, State nextState);
     '    void triggerTransition(Event event);
     '    State mCurrentState;
     '    std::vector\< std::tuple\<Action, State\> \>
       mTransitions;
     '    <iRequired> m<iRequired>;
     '};
     '";
  writeFile(oProvidedInterface, rVal);
}
private str createTransitions(sm, nameReqIface) {
  rVal = "";
  for (<c, e, a, n> <- sm) {
    rVal += "createTransition(State::S<c>,
      Event::E<split("::", e)[1]>, &<a>, State::S<n>);
            '";
  }
  return rVal;
}
private str triggerTransitions(events) {
  rVal = "";
  for (e <- events) {
    rVal += "void <e>() {
      triggerTransition(Event::E<split("::", e)[1]>); }
            '";
  }
  return rVal;
}
...
```

Listing 18. Implementation Generator in Rascal

References

[1] Rodin Aarssen. 2017. cwi-swat/clair: v0.1.0. (2017). https://doi.org/10.5281/zenodo.891122

[2] Rodin Aarssen and Tijs van der Storm. 2020. High-fidelity metaprogramming with separator syntax trees. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 27–37.

[3] Mansi Agnihotri and Anuradha Chug. 2020. A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems* 16, 4 (2020), 915–934.

[4] Omar al Duhaiby, Arjan Mooij, Hans van Wezep, and Jan Friso Groote. 2018. 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, Proceedings, Part IV 8*. Springer, 121–138.

[5] Shahbaz Ali, Hailong Sun, and Yongwang Zhao. 2021. Model learning: a survey of foundations, tools and applications. *Frontiers of Computer Science* 15 (2021).

[6] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.

[7] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. 2003. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 65–74.

[8] Oliver Biggar, Mohammad Zamani, and Iman Shames. 2021. Modular Decomposition of Hierarchical Finite State Machines. *arXiv preprint arXiv:2111.04902* (2021).

[9] Stephen D Brookes and AW Roscoe. 2021. CSP: A practical process algebra. In *Theories of Programming: The Life and Works of Tony Hoare*. 187–222.

[10] Francesco Calzolai, Rocco De Nicola, Michele Loreti, and Francesco Tiezzi. 2008. TAPAs: A tool for the analysis of process algebras. *Transactions on Petri Nets and Other Models of Concurrency I* (2008), 54–70.

[11] Joost Engelfriet. 1985. Determinancy→(observation equivalence= trace equivalence). *Theoretical Computer Science* 36 (1985), 21–25.

[12] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

[14] Emden Gansner, Eleftherios Koutsofios, and Stephen North. 2006. *Drawing graphs with dot*. Technical Report. AT&T Research.

[15] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. 2013. CADP 2011: a toolbox for the construction and analysis of

distributed processes. *International Journal on Software Tools for Technology Transfer* 15, 2 (2013), 89–107.

[16] Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modelling and analysis of communicating systems*. MIT press.

[17] David Harel and Hillel Kugler. 2004. The Rhapsody semantics of statecharts (or, on the executable core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering*. Springer, 325–354.

[18] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. 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, 307–322.

[19] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The open-source LearnLib. In *International Conference on Computer Aided Verification*. Springer, 487–495.

[20] James Ivers, Ipek Ozkaya, Robert L Nord, and Chris Seifried. 2020. 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*. 1521–1524.

[21] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. 2009. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 168–177.

[22] Kenneth Krohn and John Rhodes. 1965. Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. *Trans. Amer. Math. Soc.* 116 (1965), 450–464.

[23] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610.

[24] Daniel Neider, Rick Smetsers, Frits Vaandrager, and Harco Kuppens. 2019. Benchmarks for automata learning and conformance testing. *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday* (2019), 390–416.

[25] Ammar Osaiweran, Mathijs Schuts, Jozef Hooman, Jan Friso Groote, and Bart van Rijnsoever. 2016. Evaluating the effect of a lightweight formal technique in industry. *International Journal on Software Tools for Technology Transfer* 18, 1 (2016), 93–108.

[26] Colin Potts. 1993. Software-engineering research revisited. *IEEE software* 10, 5 (1993), 19–28.

[27] Bernhard Rumpe. 2016. *Modeling with UML*. Springer.

[28] Raghvinder S Sangwan and Colin J Neill. 2009. Characterizing essential and incidental complexity in software architectures. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. IEEE, 265–268.

[29] Mathijs Schuts, Rodin Aarssen, Paul Tielemans, and Jurgen Vinju. 2022. Large-scale Semi-automated Migration of Legacy C/C++ Test Code. *Software: Practice and Experience* (2022).

[30] Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. 2016. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *International Conference on Integrated Formal Methods*. Springer, 311–325.

[31] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N Jansen. 2015. Applying automata learning to embedded control software. In *International Conference on Formal Engineering Methods*. Springer, 67–83.

[32] Frits Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95.

[33] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. 2022. A new approach for active automata learning based on apartness. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Proceedings, Part I*. Springer, 223–243.

[34] Rutger van Beusekom, Bert de Jonge, Paul Hoogendijk, and Jan Nieuwenhuizen. 2021. Dezyne: Paving the way to practical formal software engineering. *arXiv preprint arXiv:2108.02962* (2021).

[35] Andrzej Wąsowski and Peter Sestoft. 2002. *On the formal semantics of visualSTATE statecharts*. Technical Report. TR-2002-19, IT University of Copenhagen.

Received 2023-04-28; accepted 2023-08-11