Extracting High-Level System Specifications from Source Code via Abstract State Machines

Flavio Ferrarotti(✉), Josef Pichler, Michael Moser, and Georg Buchgeher

Software Competence Center Hagenberg, Hagenberg, Austria
{flavio.ferrarotti,josef.pichler,michael.moser,georg.buchgeher}@scch.at

Abstract. We are interested in specifications which provide a consistent high-level view of systems. They should abstract irrelevant details and provide a precise and complete description of the behaviour of the system. This view of software specification can naturally be expressed by means of Gurevich's Abstract State Machines (ASMs). There are many known benefits of such an approach to system specifications for software engineering and testing. In practice however, such specifications are rarely generated and/or maintained during software development. Addressing this problem, we present an exploratory study on (semi) automated extraction of high-level software specifications by means of ASMs. We describe, in the form of examples, an abstraction process which starts by extracting an initial ground-level ASM specification from Java source code (with the same core functionality), and ends in a high-level ASM specification at the desired level of abstraction. We argue that this process can be done in a (semi) automated way, resulting in a valuable tool to improve the current software engineering practices.

1 Introduction

We consider good software specifications to be much more than just prototypes to build systems. In our view, they should also enable us to explore, reuse, debug, document and test systems, and to explain their construction in a verifiable way. In particular if these specification (models) are meant to help practitioners to manage complex software-intensive systems.

There are many formal and semi-formal software specification methods which realize this view, have been around for many year, and have successfully been applied in practice. See [21] for an overview with practical focus of the main methods, including ASM, UML, Z, TLA+, B and Estelle among others.

The research reported in this paper results from the project *Higher-Order Logics and Structures* supported by the Austrian Science Fund (FWF: [**I2420-N31**]). It was further supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

K.-D. Schewe and N. K. Singh (Eds.): MEDI 2019, LNCS 11815, pp. 267–283, 2019.
https://doi.org/10.1007/978-3-030-32065-2_19

Software specifications should serve both, the client and the developer. The client should be able to understand the specification so that she/he can validate it. The developer should have as much freedom as possible to find the best implementation that conforms to the specification. Thus, it is fundamentally important for a good specification to present a consistent high-level description of the system abstracting away irrelevant details. Let us illustrate this point with an example.

Example 1. Suppose we are given the task of specifying an algorithm for sorting a sequence of elements in-place. The algorithm must proceed sequentially, making exactly one swap in each step until the sequence is in order. The abstract state machine (ASM) in Listing 1.1 provides a formal, yet high-level specification of such algorithm. Provided we are aware that lines 3 and 4 are executed in parallel and thus there is no need to save the value of $array(i)$ into a temporary variable, this formal specification should be self-explanatory. It is not the most efficient algorithm for the task at hand, but it is the most general and gives freedom to the developer to refine it into an implementation of her/his choice such as bubble sort, insertion sort or Shell sort, among others.

```
1 rule $Sort$ =
2 choose $i, j \in indices(array)$ with $i < j$ and $array(i) > array(j)$
3             $array(i) := array(j)$
4             $array(j) := array(i)$
```

Listing 1.1. ASM High-Level Specification of Sort Algorithm.

Despite the many benefits that good high-level formal software specifications bring to software design, verification by reasoning techniques, validation by simulation and testing, and documentation, they are still a rare occurrence in the software industry, except in the case of mission critical systems. The commonly cited problems for their adoption in practice are the restrictive time and money constraints under which software is developed, and the dynamic nature of software evolution which makes difficult to keep design and documentation up to date. The need for good software specifications is further underlined by the fact that most programmers need to work on software which was not designed or developed by them, and the growing demand to document and reimplement legacy software systems.

In order to alleviate these problems, since early on [12] a considerable amount of effort have been put into reverse engineering higher level abstractions from existing systems. Nowadays the common approach in the literature (see for instance [22]) is to transform a given program P which conforms to a given grammar G into a high-level model M which conforms to a meta model MM. During this transformation P is usually represented by an abstract syntax tree or a concrete syntax tree. The extraction process relies in the specification of mappings between elements of G and MM. Independently for the specific details, this transformation is done in one big step from P to M, and the level of abstraction of M is fixed, determined by the mappings from G to MM.

In this paper we propose a different approach. Instead of relying on a big step transformation from the source code to a model at the desired level of abstraction, we propose to derive formal software specifications by a sequence of (semi) automated transformations, in which each transformation increases the level of abstraction of the previous specification in the sequence. We argue that this process can be done in a (semi) automated way and thus result in a valuable tool to improve the current software (reverse) engineering practices.

The method for high-level system design and analysis known as the ASM method [9] inspired our idea of extracting high-level specifications from software following an organic and effectively maintainable sequence of rigorous and coherent specifications at stepwise higher abstraction levels. The simple but key observation to this regard is that the process of stepwise refinement from high-level specification down to implementation provided by the ASM method, can be applied in reverse order and thus used for the (semi) automated extraction of high-level software specifications from source code.

The idea of using formal methods to reverse engineering formal specifications is of course not new. Already in the nineties, Z notation was used to extract formal specifications from COBOL [11,23]. Declarative specifications such as Z imply a fixed level of abstraction for design and verification, completely independent of any idea of computation. This is unsuitable for the multi-step abstraction approach proposed in this paper. In this sense, ASMs provide us with the required unifying view of models of computation and declarative frameworks [5].

The paper is organized as follows. In Sect. 2 we argue why ASMs provide the correct foundations for the method presented in this work for high-level software specification extraction. The actual method for stepwise abstraction of software specifications is presented in Sect. 3. In Sect. 4 we show how the method works in practice through a complete example. We conclude our work in Sect. 5.

2 Abstract State Machines

A distinctive feature of the ASM method which is not shared by other formal and semi-formal specification methods such as B, Event-B and UML is that, by the ASM thesis (first stated in [18,19] as project idea for a generalization of Turing's thesis), ASMs can step-by-step faithfully model algorithms at *any level of abstraction*. This thesis has been theoretically confirmed for most well known classes of algorithms, including sequential [20], parallel [2,3,14], concurrent [8], reflective [13], and even quantum [16] algorithms. Moreover, it has long been confirmed in practice (see [4] and Chapter 9 in [9] for a survey). This distinctive feature is a key component of the approach that we propose in this paper to extract specifications from source code. Moreover, ASMs provide simple foundations and a uniform conceptual framework (see Section 7.1 in [9]).

This paper can be understood correctly by reading our ASM rules as pseudocode over abstract data types. Nevertheless, we review some of the basic ASM features in order to make the paper self-contained. The standard reference book for this area is [9].

The *states* of ASMs are formed by a domain of elements or objects and a set of functions defined over this domain. That is, states are arbitrary universal structures. Predicates are just treated as characteristic functions. The collection of the types of the functions (and predicates) which can occur in a given ASM is called its *signature*.

In its simplest form an ASM of some signature Σ can be defined as a finite set of transition rules of the form **if** *Condition* **then** *Updates* which transforms states. The condition or guard under which a rule is applied is an arbitrary first-order logic sentence of signature Σ. *Updates* is a finite set of assignments of the form $f(t_1, \ldots, t_n) := t_0$ which are executed in parallel. The execution of $f(t_1, \ldots, t_n) := t_0$ in a given state S proceeds as follows: first all parameters $t_0, t_1, \ldots, t_n$ are assigned their values, say $a_0, a_1, \ldots, a_n$, then the value of $f(a_1, \ldots, a_n)$ is updated to a_0, which represents the value of $f(a_1, \ldots, a_n)$ in the next state. Such pairs of a function name f, which is fixed by the signature, and optional argument $(a_1, \ldots, a_n)$ of dynamic parameters values a_i, are called locations. They represent the abstract ASM concept of memory units which abstracts from particular memory addressing. Location value pairs (l, a), where l is a location and a is a value, are called updates and represent the basic units of state change.

The notion of ASM run (or equivalently computation) is an instance of the classical notion of the computation of transition systems. An ASM computation step in a given state consists in executing simultaneously all updates of all transition rules whose guard is true in the state. If these updates are consistent, the result of their execution yields a next state, otherwise it does not. A set of updates is consistent if it contains no pairs $(l, a), (l, b)$ of updates to a same location l with $a \neq b$. Simultaneous execution, as obtained in one step through the execution of a set of updates, provides a useful instrument for high-level design to locally describe a global state change. This synchronous parallelism is further enhanced by the transition rule **forall** x **with** φ **do** r which expresses the simultaneous execution of a rule r for each x satisfying a given condition φ. Similarly, non-determinism as a convenient way of abstracting from details of scheduling of rule executions can be expressed by the rule **choose** x **with** φ **do** r, which means that r should be executed with an arbitrary x chosen among those satisfying the property φ.

3 The Stepwise Abstraction Method

In this section we present a stepwise abstraction method to extract high-level specifications from source code. The method comprises the following two phases:

1. *Ground specification extraction:* This is the first step consisting on parsing the source code of the system in order to translate it into a behaviourally equivalent ASM. Here we use the term *behaviourally equivalent* in the precise sense of the ASM thesis (see [2,3,8,13,14,20] among others), i.e., in the sense that behaviourally equivalent algorithms have step-by-step exactly the

same runs. Thus the ground specification is expected to have the same core functionality as the implemented system.

2. *Iterative high-level specification extraction:* After the first phase is completed, the ground ASM specification is used as a base to extract higher-level specifications, by means of a semi-automated iterative process. The implementation of the method must at this point present the user with different options to abstract away ASM rules and/or data.

A detailed analysis of these phases follows.

3.1 Ground Specification Extraction

In this phase we focus on how to extract an ASM behaviourally equivalent to a source code implementation in a given programming language. In our research center, we have a positive industrial experience in parsing and extracting knowledge from source code[1]. We have built and applied several reverse engineering tools around the Abstract Syntax Tree Metamodel (ASTM) standard[2]. In particular, we have shown its potential for multi-language reverse engineering in practice [15].

Using this approach and adapting our previous results, we have determined that it is possible extract the desired ground specifications in the form of a behaviourally equivalent ASM by automated means. The idea is to transform the source code into an ASM model in two steps. First, by means of eKnows[3], the source code is parsed into a language-agnostic canonical AST representation. Besides concrete language syntax, this intermediate representation also abstracts from language-specific semantics with regard to control-flow. For instance, the switch statement occurs in different forms, namely with or without fall-through semantics. eKnows constructs an AST representation with a standardized semantic (e.g. explicit break statements even for non-fall-through languages) that allows homogeneous subsequent analysis/transformation steps. Furthermore, eKnows resolves unstructured control-flow (e.g. break and continue within loop statements, or goto statement) by means of refactoring resulting in well-structured control-flow, i.e. single entry/exit points of statements.

In the second step, we provide rewriting rules for AST nodes specifically related to control-flow (e.g. for loops, conditional statements) and assignment statements. Rewriting rules for control-flow nodes can be applied in a straightforward way to individual nodes independent of any context information. The subsequent examples illustrate the idea for the transformation of loop statements. The transformation of assignment statements, however, need semantic analysis of the source code due to the difference between strict sequential execution order of program code and simultaneous execution of ASM update rules. We can leverage symbolic execution (also part of eKnows) in order to eliminate intermediate variables and construct assignment statements that only contain

[1] http://codeanalytics.scch.at/.
[2] https://www.omg.org/spec/ASTM/1.0/.
[3] https://www.scch.at/de/eknows.

input/output parameters of the analyzed algorithms. In this transformed representation, the strict execution sequence becomes irrelevant and statements can be transformed into behaviorally equivalent ASM update rules executed in parallel.

Next we present a simple example of ground level ASM specifications extracted from a Java implementation of the bubble sort algorithm.

Example 2. Let us analyse the very simple and compact bubble sort algorithm implemented by the Java method in Listing 1.2.

```
1 public static void bubbleSort(int array[]) {
2     for (int n = array.length - 1; n > 0; n--) {
3         for (int i = 0; i < n; i++) {
4             if (array[i] > array[i+1]) {
5                 int temp = array[i];
6                 array[i] = array[i+1];
7                 array[i+1] = temp;} } } }
```

Listing 1.2. Bubble sort algorithm as Java method.

Let `for` be the following iterative turbo ASM rule which first executes the rule R_0, and then repeats the execution of its body rule R_2 followed by R_1 as long as they produce a non-empty update set and the condition *cond* holds.

```
for ($R_0$; $cond$; $R_1$) $R_2$ =
                        $R_0$ seq iterate (if $cond$ then $R_2$ seq
    $R_1$)
```

The turbo ASM in Listing 1.3 can easily be obtained from the Java code in Listing 1.2 by mostly simple syntactic rewriting, except for the value swap done in parallel in lines 5 and 6 which requires a simple semantic abstraction of lines 5–7 in Listing 1.2. Using the symbolic approach described above, the Java variable `temp` in the assignment in line 7 of Listing 1.2 would get substituted by the previous assignment (line 5) resulting in the ASM rule $array(i+1) := array(i)$.

```
1 rule $bubbleSort0$ =
2     for ($n := array.length - 1$; n > 0; $n := n - 1$)
3         for ($i := 0$; $i < n$; $i := i + 1$)
4             if $array(i) > array(i+1)$ then
5                 $array(i) := array(i+1)$
6                 $array(i+1) := array(i)$
```

Listing 1.3. Turbo ASM extracted from Java method `bubbleSort`.

Alternatively, we can extract from Listing 1.2 the control state ASM in Listing 1.4. Same as in the case of the turbo ASM, the transformation from the Java method `bubbleSort` to the control state ASM only requires simple rewriting techniques.

```
rule $bubbleSort1$ =
    if $state = s_0$ then
        $n := array.length - 1$
        $state := s_1$
    if $state = s_1$ then
        if $n > 0$ then
            $i := 0$
            $state := s_2$
    if $state = s_2$ then
        if $i < n$ then
            if $array(i) > array(i+1)$ then
                $array(i) := array(i+1)$
                $array(i+1) := array(i)$
            $i := i + 1$
        else
            $n := n - 1$
            $state := s_1$
```

Listing 1.4. Control State ASM abstracted from Java code of `bubbleSort`.

Although the control state ASM in Listing 1.4 has more lines of code than the turbo ASM in Listing 1.3 (and that the original Java code), it has certain advantages. It can for instance be represented graphically as the UML-style diagram in Fig. 1. Furthermore, the control state ASM presents a transparent white-box view of the states while the turbo ASM presents a black-box view which hides internal sub-computations. Which of these views is more useful depends on the desired specification level and the application at hand, and can be decided by the user. For instance, at low levels of abstraction, control ASMs can lead to complex UML-style diagrams which might share the usual drawbacks of UML activity diagrams, unnecessarily replacing elegant structured code by control flow based in states (a kind of "hidden goto" if viewed as a program).

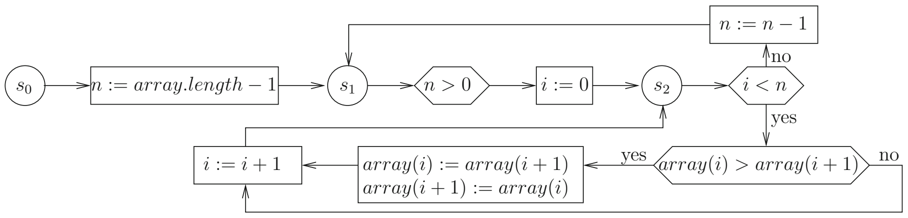


Fig. 1. bubbleSort1

3.2 Iterative High-Level Specification Extraction

Without entering into technical details, which are nevertheless well explained in Section 3.2 in [9], we note that the schema of ASM *refinement step*, can also be viewed as describing an *abstraction step* if it is used for extracting a high-level model. In this sense, the re-engineering project in [1] confirms this idea in practice and it is a source of inspiration for our proposal.

Thus, when abstracting an ASM M' from an $ASM\ M$, there is a lot of freedom. In particular, it is possible to perform diverse kinds of abstractions by combining the following items:

– The notion of an abstracted state, obtained by changing (usually reducing) the signature.
– The notion of state of interest and correspondence between states, obtained by changing (usually reducing) the states of interest in M' with respect to M, and determining the pairs of states of interest in the runs of M and M' that we want to relate through the abstraction.
– The notion of abstract computation segments, obtained by changing (usually reducing) the number of abstract steps that lead to states of interest of M' with respect to M.
– The notion of locations of interest and of corresponding locations, obtained by changing (usually reducing) the locations of interest of M', and determining the pairs of corresponding locations of interest of M and M' that we want to relate through abstraction. Recall that in ASM terminology the term location refer to abstract data container.
– The notion of equivalence of the data in the locations of interest, obtained by changing (usually reducing) the number of different classes of equivalence, and thus also changing the notion of equivalence of corresponding states of interest of M and M'.

The aim of this phase is to semi-automatically extract high-level system specifications starting from the ground ASM specification built in the previous phase. In this paper we present a proof of concept through examples. We show that very simple heuristic analyses of ASM rules can already lead to useful abstractions. Of course, much more sophisticated abstraction mechanisms such as abstract interpretations are certainly possible, opening what would constitute an interesting research project.

Example 3. Note that it is clear from the ASM ground specifications extracted in Example 2 that the order in which the values in the sequence are swapped does not really matter from a conceptual high-level perspective. That is, if we keep swapping the values of $array(i)$ and $array(i+1)$ as long as $array(i) > array(i+1)$ for some index i, then the algorithm still works correctly. We thus can abstract the ASM in Listing 1.5. After at most $2n$ steps, where n is the length of the array, we have that for every index i the condition in the choose rule no longer holds. At that point the machine stops and, for every index i, it holds that $array(i) \leq array(i+1)$, i.e., the array is in order. Clearly, the ASM in Listing 1.5 is not only an abstraction of the bubble sort algorithm, it is also an specialization of the in place sorting algorithm specified by the ASM rule in Listing 1.1.

```
rule bubbleSort2 =
    choose $i$ with $0 \le i < array.length - 1$ and $array(i) > array(i+1)$
        $array(i) := array(i+1)$
        $array(i+1) := array(i)$
```

Listing 1.5. Abstraction from control state ASM *bubbleSort*1.

Admittedly, high-level specifications such as the one in Listing 1.5 are not trivial to abstract following mechanical procedures, since they are not decidable in the general case. We can however analyse and classify programming patterns, applying engineering and AI techniques such as heuristics, symbolic execution, machine learning, theorem provers etc. to identify appropriate and correct abstractions of this kind.

4 Dijkstra's Shortest Path Algorithm: Extracting High-Level Specifications from a Java Implementation

In this section we showcase a step-by-step *formal* process of abstraction from a Java implementation of the famous Dijkstra's shortest path algorithm, up to a high-level ASM specification of the underpinning graph traversal algorithm. The correctness of each step of this abstraction process can be formally proven following similar arguments to those in the refinement proofs of Sect. 3.2 of the ASM book [9]. Automated proving would also be possible in some cases, but that is not the focus of this work.

We start from a Java implementation (slightly adapted from the one in https://www.baeldung.com/java-dijkstra) of the shortest path algorithm. Rather surprisingly we show that applying our method we can extract very similar high-level specifications to those in Sect. 3.2 of the ASM book [9].

In the Java implementation of the algorithm graph are represented as sets of nodes. Each node is an object of the class in Listing 1.6 which has a name, an upper bound for its distance from source, and a list of adjacent nodes.

```java
public class Node {
    private String name;
    private Integer upbd = Integer.MAX_VALUE;
    private Map<Node, Integer> adj = new HashMap<>();
    public Node(String name) {
        this.name = name;
    }
    public void addDestination(Node destination, int
   weight) {
        adjacentNodes.put(destination, weight);
    }
    // getters and setters ...
}
```

Listing 1.6. Class Node.

It is not difficult to see that the Java method in Listing 1.7 actually implements Dijkstra's shortest path algorithm. This is the case mainly because: (a) the algorithm is well known, (b) the implementation is quite standard, and (c) the code is quite short. If either of (a), (b) or (c) does not hold, then the task of understanding the program would certainly be more time consuming and challenging.

```java
public static Graph shortestPath(Graph graph,Node source)
   {source.setUpbd(0);
    Set<Node> visited = new HashSet<>();
    Set<Node> frontier = new HashSet<>();
    frontier.add(source);
    while (frontier.size() != 0) {
        Node u = LowestDistanceNode(frontier);
        frontier.remove(u);
        for (Entry<Node,Integer> pair :
        u.getAdj().entrySet()) {
            Node v = pair.getKey();
            Integer weight = pair.getValue();
            LowerUpbd(u,v,weight);
            if (!visited.contains(v)) {
                visited.add(v);
                frontier.add(v);}}}
    return graph;}

private static void LowerUpbd(Node u, Node v, Integer
    weight) {
    if (u.getUpbd() + weight < v.getUpbd()) {
        v.setUpbd(u.getUpbd() + weight);}}
```

Listing 1.7. Shortest path algorithm as Java program.

Let us now abstract the code of `shortestPath` by transforming it into a control state ASM. We follow a very simple procedure which consists mostly on syntactic rewriting. First we note that we can represent the omnipresent binding or instantiation of methods and operations to given objects, by means of parametrized functions [7]. The schema can be expressed by the equation $\mathit{self}.f(x) = f(\mathit{self}, x)$ or $f(x) = f(\mathit{self}, x)$. The parameter *self* is used to denote an agent, typically the one currently executing a given machine. This is similar to the object-oriented current class instance *this* with respect to which methods of that class are called (executed). In an object oriented spirit the parameter *self* is often left implicit.

The state in which the control state ASM will operate is easily abstracted from the input parameters to the method `shortestPath`, i.e., the `Graph` and `Node` classes. We omit a detailed description here since it will be clear from the context.

The rewriting of `shortestPath` into a *behavioural equivalent* ASM control state machine *ShortestPath* proceeds as follow:

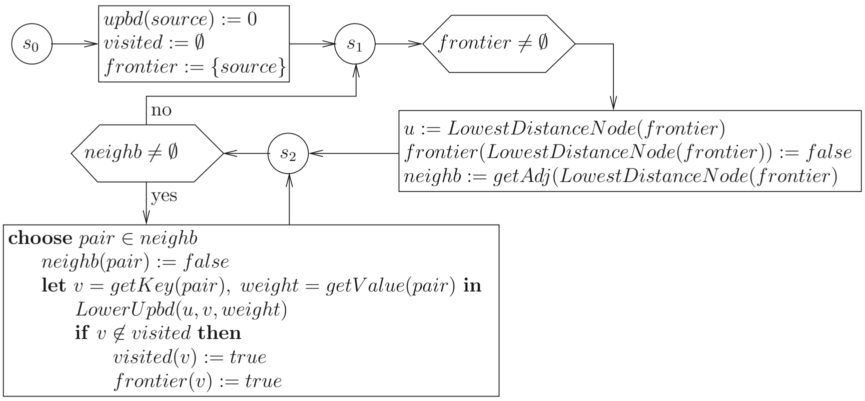
1. Lines 2–4 in Listing 1.7 translate to simple updates of the current upper bound value of the node `source` to 0 (initially set to `Integer.MAX_VALUE`, or ∞ in ASM notation) and the values of `visited` and `frontier` to empty sets. These three updates can be done in parallel and thus in the initial control state s_0. The update to `frontier` in Line 5 cannot be done in parallel with that of Line 4. Nevertheless, a simple heuristic can easily discover that these two updates can be collapsed into one. Thus Lines 2–5 can be translated to the parallel updates shown in lines 3–5 in Listing 1.8.
2. The while-loop starting in line 6 requires a new control state to which the ASM can return. If we are in this state and the condition in the while-loop is satisfied, then the rule corresponding to the code inside the while-loop is applied and the machine remains in control state s_1. See lines 3–5 in Listing 1.8.
3. Lines 7–8 can be done in parallel since they require updates to different locations. See lines 9–10 in Listing 1.8.
4. Same as the while-loop, the for-loop in line 9 calls for a new control state and to keep track of the nodes adjacent to `LowestDistanceNode(frontier)` which have not been visited yet. Once the for-loop is done, i.e., there is no more adjacent nodes to visit, we need to return to the control state s_1, since this for-loop is nested in the while-loop being represented in that control state. The result is shown in lines 11–16 and 22–23 of Listing 1.8.
5. The values of v and *weight* are only defined and used locally in lines 10–15. In addition, lines 12–15 update disjoint state locations, and there is no interdependency among them. Thus, we can replace lines 10–11 by a let-rule and process lines 12–15 in parallel. See lines 17–21 in Listing 1.8.
6. Finally, the ASM rule *LowerUpbd* is almost identical to the `LowerUpbd` method, except for the trivial differences in notation. In *LowerUpbd*, parameter u is a location variable while parameters v and *weight* are logical variables. See lines 25–27 in Listing 1.8.

```
1  rule $ShortestPath0$ =
2      if $state = s_0$ then
3          $upbd(source) := 0$
4          $visited := \emptyset$
5          $frontier := \{source\}$
6          $state := s_1$
7      if $state = s_1$ then
8          if $frontier \neq \emptyset$ then
9              $u := LowestDistanceNode(frontier)$
10             $frontier(LowestDistanceNode(frontier)) := false$
11             $neighb := getAdj(LowestDistanceNode(frontier))$
12             $state := s_2$
13     if $state = s_2$ then
```

```
14           if $neighb \neq \emptyset$ then
15               choose $pair \in neighb$
16                   $neighb(pair) := false$
17                   let $v = getKey(pair),\ weight = getValue(pair)$ in
18                       $LowerUpbd(u, v, weight)$
19                       if $v \notin visited$ then
20                           $visited(v) := true$
21                           $frontier(v) := true$
22           else
23               $state := s_1$
24
25   rule $LowerUpbd(u, v, weight) =$
26       if $upbd(u) + weight < upbd(v)$ then
27           $upbd(v) := upbd(u) + weight$
```

Listing 1.8. Control State ASM extracted from the Java code of `shortestPath`.

Being *ShortestPath*0 a control state ASM, we can represent it using UML-style graphical notation. This gives us the self explanatory Fig. 2.



s_0
$upbd(source) := 0$
$visited := \emptyset$
$frontier := \{source\}$
s_1
$frontier \neq \emptyset$
no
$u := LowestDistanceNode(frontier)$
$frontier(LowestDistanceNode(frontier)) := false$
$neighb := getAdj(LowestDistanceNode(frontier))$
$neighb \neq \emptyset$
s_2
yes
choose $pair \in neighb$
$neighb(pair) := false$
let $v = getKey(pair),\ weight = getValue(pair)$ in
$LowerUpbd(u, v, weight)$
if $v \notin visited$ then
$visited(v) := true$
$frontier(v) := true$

Fig. 2. ShortestPath0

Examining the code in lines 15–21 in Listing 1.8 plus the rule *LowerUpbd*, it is not difficult to conclude that instead of extending the frontier by one neighbour of u at a time, we can extend it as a wave, i.e., in one step we can in parallel extend the frontier to all neighbours of u. This is so because the choose rule that select the neighbour to be processes in each round, implies that the order in which this is done does not affect the result. Furthermore, there is no possibility of clashes since the updated locations $visited(v)$, $frontier(v)$ and $upbd(v)$ are all disjoint for different values of v. Thus we can abstract from *ShortestPath*0 the control state ASM *ShortestPath*1 in Listing 1.9 , where we replace the choose rule by a for all rule.

```
rule $ShortestPath1$ =
    if $state = s_0$ then
        $upbd(source) := 0$
        $visited := \emptyset$
        $frontier := \{source\}$
        $state := s_1$
    if $state = s_1$ then
        if $frontier \neq \emptyset$ then
            $u := LowestDistanceNode(frontier)$
            $frontier(LowestDistanceNode(frontier)) := false$
            $neighb := getAdj(LowestDistanceNode(frontier))$
            $state := s_2$
    if $state = s_2$ then
        forall $pair \in neighb$
            let $v = getKey(pair),\ weight = getValue(pair)$ in
                $LowerUpbd(u, v, weight)$
                if $v \notin visited$ then
                    $visited(v) := true$
                    $frontier(v) := true$
        $state := s_1$
```

Listing 1.9. Abstraction of the *ShortestPath*0 ASM rule.

As a next step, we can simply eliminate control state s_2 by using a let rule, changing u and $neighb$ from state locations to logical variables. The result is shown in Listing 1.10.

```
rule $ShortestPath1$ =
    if $state = s_0$ then
        $upbd(source) := 0$
        $visited := \emptyset$
        $frontier := \{source\}$
        $state := s_1$
    if $state = s_1$ then
        if $frontier \neq \emptyset$ then
            let $u = LowestDistanceNode(frontier)$ ,

   $neighb = getAdj(LowestDistanceNode(frontier))$
            in $frontier(u) := false$
                forall $pair \in neighb$
                    let
   $v = getKey(pair),\ weight = getValue(pair)$ in
                        $LowerUpbd(u, v, weight)$
                        if $\notin visited(v)$ then
                            $visited(v) := true$
                            $frontier(v) := true$
```

Listing 1.10. Abstraction of the ShortestPath1 ASM rule.

At this point we have quite an abstract view of the shortest path algorithm. We can nevertheless continue this abstraction process. An interesting possibility to this regard is to eliminate the information regarding edge weights. In this way, we get the ASM in Listing 1.11. It is not difficult to see that the resulting ASM no longer calculates the shortest path from the source. It has been transformed into an ASM that specifies the graph traversal algorithm which underpins the shortest path algorithm.

```
rule $GraphrTraversal0$ =
    if $state = s_0$ then
        $upbd(source) := 0$
        $visited := \emptyset$
        $frontier := \{source\}$
        $state := s_1$
    if $state = s_1$ then
        if $frontier \neq \emptyset$ then
            choose $u \in frontier$
                $frontier(u) := false$
                forall $v \in getAdj(u)$
                    if $v \notin visited(v)$ then
                        $visited(v) := true$
                        $frontier(v) := true$
```

Listing 1.11. Abstraction of the *ShortestPath*2 ASM rule.

We can further abstract *GraphrTraversal*0 by processing all the nodes in the frontier in parallel instead of one-by-one. This is the same idea that we use to abstract *ShortestPath*1 from *ShortestPath*0. In this way, we get the ASM in Listing 1.12.

```
1  rule $GraphrTraversal1$ =
2      if $state = s_0$ then
3          $upbd(source) := 0$
4          $visited := \emptyset$
5          $frontier := \{source\}$
6          forall $u \in frontier$
7              $frontier(u) := false$
8              forall $v \in getAdj(u)$
9                  if $v \notin visited(v)$ then
10                     $visited(v) := true$
11                     $frontier(v) := true$
```

Listing 1.12. Abstraction of the *GraphTraversal*0 ASM rule.

A somehow more abstract view can still be obtained if we simply replace lines 8–10 in Listing 1.12 by a function defined by a sub-machine.

5 Conclusion

We have argued that it is possible to derive high-level formal software specifications in the form of ASMs by a sequence of (semi) automated transformations, in which each transformation increases the level of abstraction of the previous specification in the sequence. This provides a new tool to improve the current software (reverse) engineering practices as shown by the encouraging results of the small experimental examples presented in this paper. The proposed approach to software re-engineering have several advantages, including:

- Precise, yet succinct and easily understandable specifications at desired levels of abstraction.
- Each abstraction/refinement step can be proven correct if needed. This enables for instance to prove that the implementation satisfies the requirement.
- Only the first abstraction from source code to ASM rules depends from the programming language of the implementation. Subsequent abstractions only rely on general principles and transformations of ASM rules.
- The initial abstraction from source code to ASM can potentially be done entirely automatically via rewriting rules.
- Enables the exploitation of abstraction for specification reuse.
- Specifications are executable for instance in CoreASM or Asmeta.
- Can be used for reverse engineering/understanding (legacy) source code.
- Can be used to produce finite state machines for model based testing. For instance by means of refinement of the extracted high-level ASM models to finite state machines [17].
- Interactive exploration of the design on all abstraction levels, enabling the discovery of high-level bugs.

The natural next step is to confirm the observations in this paper within the context of large software implementations, in the style of [1], but using our semi-automated approach instead. For that, we aim to extend our eKnows[4] platform to extract ground ASM specification from source code and experiment with software systems of our industrial partners. In parallel, we plan to carry on a systematic study of heuristics for the automated extraction of high-level ASM specifications, starting from detailed ground ASM specifications. In this sense, references [6,10] are a good starting point.

References

1. Barnett, M., Börger, E., Gurevich, Y., Schulte, W., Veanes, M.: Using abstract state machines at microsoft: a case study. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 367–379. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44518-8_21

[4] https://www.scch.at/de/eknows.

2. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Trans. Comput. Logic **4**(4), 578–651 (2003)
3. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms: correction and extension. ACM Trans. Comput. Logic **9**(3), 19:1–19:32 (2008)
4. Börger, E.: The origins and the development of the ASM method for high level system design and analysis. J. UCS **8**(1), 2–74 (2002)
5. Börger, E.: Abstract state machines: a unifying view of models of computation and of system design frameworks. Ann. Pure Appl. Logic **133**(1–3), 149–171 (2005)
6. Börger, E.: Design pattern abstractions and abstract state machines. In: Proceedings of the 12th International Workshop on Abstract State Machines, ASM 2005, 8–11 March 2005, Paris, France, pp. 91–100 (2005). http://www.univ-paris12.fr/lacl/dima/asm05/DesignPattern.ps
7. Börger, E., Cisternino, A., Gervasi, V.: Ambient abstract state machines with applications. J. Comput. Syst. Sci. **78**(3), 939–959 (2012)
8. Börger, E., Schewe, K.: Concurrent abstract state machines. Acta Inf. **53**(5), 469–492 (2016)
9. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
10. Börger, E., Stärk, R.F.: Exploiting abstraction for specification reuse. the Java/C# case study. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 42–76. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30101-1_3
11. Bowen, J.P., Breuer, P.T., Lano, K.: Formal specifications in software maintenance: from code to z^{++} and back again. Inf. Softw. Technol. **35**(11–12), 679–690 (1993)
12. Chikofsky, E.J., II, J.H.C.: Reverse engineering and design recovery: a taxonomy. IEEE Softw. **7**(1), 13–17 (1990)
13. Ferrarotti, F., Schewe, K.-D., Tec, L.: A behavioural theory for reflective sequential algorithms. In: Petrenko, A.K., Voronkov, A. (eds.) PSI 2017. LNCS, vol. 10742, pp. 117–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74313-4_10
14. Ferrarotti, F., Schewe, K., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing - simplified parallel ASM thesis. Theor. Comput. Sci. **649**, 25–53 (2016)
15. Fleck, G., et al.: Experience report on building ASTM based tools for multi-language reverse engineering. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, 14–18 March 2016, vol. 1, pp. 683–687 (2016)
16. Grädel, E., Nowack, A.: Quantum computing and abstract state machines. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 309–323. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36498-6_18
17. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, 22–24 July 2002, pp. 112–122. ACM (2002)
18. Gurevich, Y.: Reconsidering turing's thesis: toward more realistic semantics of programs. Technical Report CRL-TR-36-84, January 1984
19. Gurevich, Y.: A new thesis. Technical Report 85T–68-203, abstracts, American Mathematical Society (1985)
20. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Logic **1**(1), 77–111 (2000)

21. Habrias, H., Frappier, M.: Software Specification Methods. ISTE (2006)
22. Izquierdo, J.L.C., Molina, J.G.: Extracting models from source code in software modernization. Softw. Syst. Model. **13**(2), 713–734 (2014)
23. Lano, K., Breuer, P.T., Haughton, H.P.: Reverse-engineering COBOL via formal methods. J. Softw. Maintenance **5**(1), 13–35 (1993)