Extracting Mode Diagrams from Blech Code

Daniel Lucas*, Alexander Schulz-Rosengarten†,
Reinhard von Hanxleden†
Department of Computer Science, Kiel University
Kiel, Germany
*stu124145@mail.uni-kiel.de
†{als, rvh}@informatik.uni-kiel.de

Friedrich Gretz, Franz-Josef Grosch
Robert Bosch GmbH, Corporate Research
Renningen, Germany
{Friedrich.Gretz, Franz-Josef.Grosch}@de.bosch.com

***Abstract*—Software visualization tools can improve the software development process by providing a graphical overview of source code and enhancing collaboration. We here propose a concept to automatically extract mode diagrams from Blech code, an imperative synchronous programming language for embedded, reactive and safety-critical systems. Our main findings are that the visualization is helpful to understand the stateful nature of the source code and that it can enhance the collaboration between developers. It is also found, however, that a good understanding of the precise diagram semantics meaning of the diagram elements is key. Lastly, the findings indicate that preference on different labeling options is highly subjective.**

***Index Terms*—synchronous languages, state machines, mode diagrams, software visualization, reactive systems**

I. Introduction

Software development is more than the process of writing source code. It begins by recording the requirements for the desired software and continues with planning of the software architecture, its components, and its environment. Only then, actual source code is written. The process then includes the generation of documentation. It can be generated with visualization tools and languages such as UML. Creating such models can be a tedious task that takes a lot of effort and time. The purpose of documentation is to offer an overview of the software (components) and providing a help for understanding the code base in different abstraction levels. Documentation is beneficial to use for experienced developers as well as developers that just started out working on a given project.

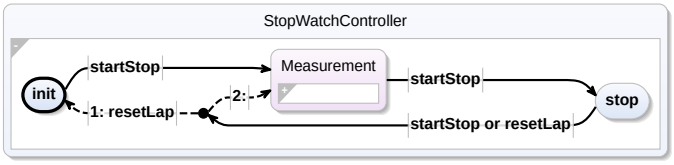
Code, however, is not static. Code improvements, refactoring, new features, customer requests or updated technology are all reasons to change existing code. Therefore, automatic generation of up-to-date code documentation could enhance the software development process.

In this paper, we investigate how to harness software visualization [6] for the Blech programming language [7]. Blech is an imperative synchronous programming language for embedded, reactive, real-time, safety-critical systems. It is a *synchronous* language [2], and is inspired by languages like Esterel [3], Céu [12] or Sequentially Constructive Statecharts (SCCharts) [16] that are used to implement stateful behavior. To illustrate that stateful nature, consider the StopWatchController example in Fig. 1. In the initial *tick* we perform some initializations and then pause at the await statement in line 10, which we can consider as the initial *state* of the program. We

978-1-6654-1825-6/21/$31.00 ©2021 IEEE

```
1  activity StopWatchController
2    (startStop: bool, resetLap: bool)   // Read–only inputs
3    (display: Display)                  // Read–write outputs
4     var totalTime: int32
5     var lastLap: int32
6     repeat
7        totalTime = 0
8        lastLap = 0
9        writeTicksToDisplay(totalTime)(display)
10       await startStop                       // State init
11       repeat
12          cobegin weak
13             await startStop
14          with weak
15             run Measurement(resetLap)(totalTime, lastLap, display)
16          end
17          writeTicksToDisplay(totalTime)(display)
18          await startStop or resetLap          // State stop
19          // Run again if only startStop was pressed
20       until resetLap end // Back to init if resetLap was pressed
21    end
22 end
```

(a) Original Blech code.



(b) The corresponding mode diagram, automatically extracted with the approach presented here.

Fig. 1: The StopWatchController example.

remain in that state until, in some future reaction, the Boolean flag startStop is true and we enter a repeat loop. In that loop, the program concurrently does two things (over several reaction steps): it waits for another occurrence of startStop, while also executing the Measurement sub-activity. After that, we wait for the startStop or resetLap flags; if the latter occurs, we loop back to the initial state.

While the underlying state machine is explicitly laid down in the program, extracting it requires some fairly careful reading of the source code. In real life, the code quickly becomes more complex than this simple example and keeping track of the stateful behavior may become a challenging task. This is where software visualization and the work presented here come into play.

Contributions and Outline

We here investigate the automatic synthesis of mode diagrams, such as Fig. 1b, from Blech code. We first present in Sec. II a *structural translation* ("Phase 1"), which takes care of directly translating Blech code into SCCharts statement by statement. This results in an SCChart that reflects the modes in the given Blech code. However, the states will not be labelled, which limits their usefulness, and the diagram will typically be quite bloated. This motivates "Phase 2," presented in Sec. III, where we propose different approaches to label mode diagrams based on Blech source code annotations, and "Phase 3," covered in Sec. IV, where we present optimization approaches to generate more concise mode diagrams. Sec. V presents evaluation results based on feedback from industrial Blech users. We wrap up with a discussion of further related work in Sec. VI and conclusions in Sec. VII.

For space considerations, we will focus here on the concepts underlying our approach. For full technical detail, we refer to the thesis of the first author [8] .

A. Background on SCCharts

Our mode diagrams are finite state machines with hierarchy and concurrency. It is not our goal to find an accurate representation of a low-level control flow graph. The term *mode* emphasizes the objective to represent somewhat higher-level modes of operation of a program. A mode may persist over several reactions and comprise an arbitrary amount of *program states* (evaluations of program counters and variables). We borrow syntax and semantics from SCCharts [16]. However, we only use a subset of SCCharts that visualizes the high-level behavior, omitting transition effects, variable declarations, etc.

To get full value from the mode diagrams, we now summarize the key syntactical SCChart elements used here. States with a thick border, as state init in Fig. 1b, are *initial states*, through which a *super state*, such as StopWatchController, is entered. States with a double border are *final states*, which denote termination of the enclosing super state. Super states consist of one or more *regions* that execute concurrently, see for example Fig. 7.

SCCharts also adopt the distinction introduced by SyncCharts [1] between *delayed* transitions, drawn with a solid line, and *immediate* transitions, drawn with a dashed line. Roughly speaking, delayed transitions denote tick/reaction boundaries, whereas immediate transitions denote control flow within a reaction. When a state is entered in a reaction, the state can be left within the same reaction only through immediate transition. E. g., in the StopWatchController, the stop state is entered when we are in the Measurement state at the beginning of the tick and the startStop *event* is *present*. Since stop does not have any outgoing immediate transitions, we end the reaction in that state. To leave the state we require another event startStop in a subsequent reaction, or an event resetLap. *Connectors*, drawn as a small black disk, are states that are guaranteed to be left again in the tick they are entered. Transitions leaving connectors must be immediate, and there must be a *default transtion* that is taken unconditionally. If a state or connector has multiple outgoing transtions, these are checked in order of their *priority*, where 1 is the highest priority. Coming back to the StopWatchController in Fig. 1, we observe that if we are in stop at the beginning of the tick, we advance to init if resetLap is present, and we transition to Measurement (via the default transition from the connector) if resetLap is present.

Finally, SCCharts also offers different types for transitions that leave super states. *Termination transitions*, indicated with a green triangle, are taken when all regions in a super state have reached a final state. *Strong abort transtions*, indicated with a red circle, suppress execution of the inner behavior of the super state when they are taken. Transitions out of a super state without any adornments are *weak abort transtions*, which permit inner behavior (a "last wish") of the left super state. Strong aborts and termination transitions are used for example to translate the when-abort statement, as shown in Fig. 9.

In addition to these pre-existing SCCharts language features, our mode diagrams also include *final regions*, indicated by a double outline, as illustrated in Fig. 7. Final regions effectively make each of their states final, which allows to capture weak branches of the Blech cobegin statement, see also Sec. II-F.

II. Phase 1: Structural Translation

Generally, the translation rules are applied recursively. The body of a hierarchical Blech element will be translated before the rest of the list on the same hierarchy level is evaluated. Unnecessary hierarchies originating from this procedure, as well as hierarchies that contain no stateful behavior, will be simplified in Phase 3.

Some translation rules append an empty *exit* state to the chart representing a Blech construct. This additional exit state serves as a connection point when combining two charts in sequence, for instance. The connecting transition emanates from the designated exit state and therefore does not interfere with transitions that are relevant for the inner behavior of the first state chart in the sequence. As before, unnecessary exit states will be removed in Phase 3.

A. Activities

In Blech, an *activity* is an encapsulated piece of code that contains reactive behavior with at least one await statement. Besides standard sequential composition, behaviors can be composed concurrently using cobegin and hierarchically using run statements. This differs from *functions* in Blech which must not contain stateful behavior and are used to encapsulate sequential computation that finishes immediately within a tick. Consequently, functions are ignored by our synthesis.

The designated entry point activity in the Blech source code will always be the outermost part of our visualization, unless the user selects another activity to start the visualization from.

Blech activities declare two list of parameters: read-only inputs and read-write outputs. This significantly simplifies the readability of interfaces and allows for a simple causality analysis. For our visualization parameters, that distinction

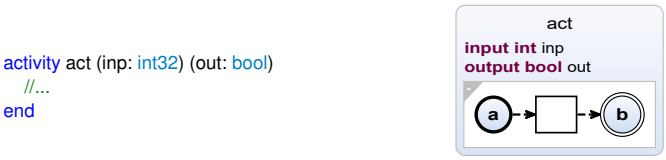


Fig. 2: Blech activity and synthesized SCChart.

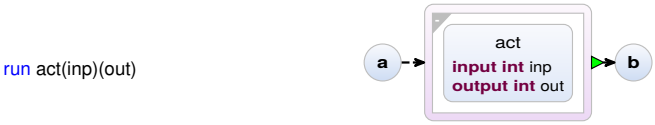


Fig. 3: Blech run statement and synthesized SCChart.

Fig. 4: Blech await statement and synthesized SCChart.

could be ignored. However, parameters may occur in expressions guarding mode transitions. In order to help the user to distinguish local activity variables from parameters we preserve the declared parameters, as shown in Fig. 2.

An activity is visualized and initialized as follows.

1) Create an empty SCChart with the name of the activity.
2) Add input and output variables of the activity as the input and output variables of the SCChart using the same names and types.
3) Add an initial state and a final state.
4) Translate the activity's body and add the result to the body of the current SCChart.
5) Connect the final state of the inner (body) SCChart to the final state of the current chart, if it exists. Otherwise (if the inner SCChart corresponds to an infinite behavior without an exit state) remove the obsolete final state from the current SCChart.

An example of the synthesis for an activity is given by Fig. 2. The Blech code contains a comment that abstracts away the code of the body. The abstracted inner behavior is the square that is connected to the initial state a and final state b via immediate transitions. The name of the activity is preserved, as well as the input and output variables.

The names of the states a and b are added here for easier description of the figures. Those labels will not be added automatically through the synthesis. However, it is possible to add custom labels for states in Phase 2, as will be discussed in Sec. III.

Now that the translation of activities has been established, we consider the statements in their bodies.

B. Run

The run statement calls one Blech activity from another. For the sake of brevity, we skip the step-by-step translation rules and simply illustrate the translation in Fig. 3.

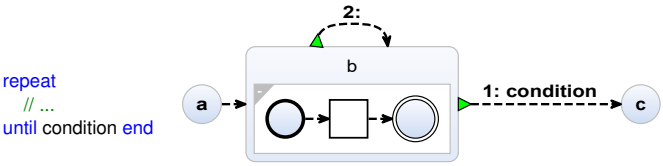


Fig. 5: Blech repeat statement and synthesized SCChart.

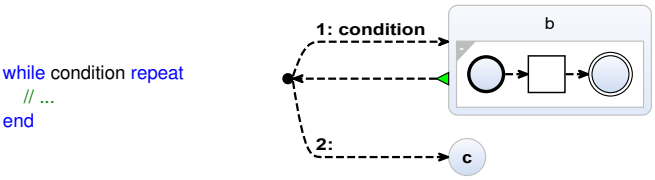


Fig. 6: Blech while statement and synthesized SCChart.

C. Await

The await statement ends the current reaction of a thread. In the next tick, the thread will resume execution at this point, check the awaited condition and, if it holds, proceed execution. In the context of this work, await is the most important statement as it clearly defines the end of a state and a transition to the next one. Fig. 4 shows the translation of an await statement.

D. Repeat

A repeat loop executes its body and then exits if the condition is true, or restarts otherwise. Fig. 5 illustrates the translation rule. Optionally, the until condition can be left out, which then yields an infinite loop.

E. While

The while construct is very similar to the repeat construct just described. The main difference in the workflow is that the condition is checked at the beginning of the loop. The loop-starting state, which was considered to be the previous state in previous translation steps, therefore has a transition into a complex state containing the body of the loop and to a state that was newly added and is the starting point for connecting subsequent statements. If the previous state is simple, it simply serves as a connection point for transitions and is not really a state. If this is the case, the previous node can be replaced by a connector (see Sec. I-A), as shown in Fig. 6.

F. Cobegin

A cobegin statement represents concurrency. Concurrent branches can be *weak*, which means they are weakly aborted by other branches that have terminated. For example, an infinite loop in a weak branch can be aborted by a simple await statement in a concurrent branch. The awaiting branch terminates on a given condition and then aborts the weak branch. When aborted, a weak branch will continue to execute until the current tick ends, for example with an await statement.

```
cobegin weak
   // ..
with
   // ..
end
```

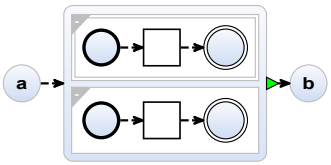


Fig. 7: Blech cobegin statement, with a weak and a strong branch, and synthesized SCChart. The weak branch is translated to a final region, indicated by a double outline.

```
if condition then
   // ..
else
   // ..
end
```

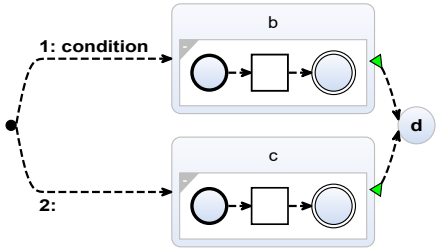


Fig. 8: Blech if-else statement and synthesized SCChart. The initial state is transformed into a connector state.

```
when condition
      abort
   // ..
end
```

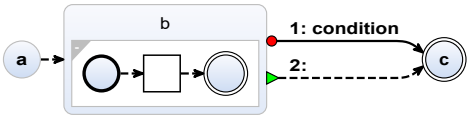


Fig. 9: Blech when-abort statement and synthesized SCChart.

```
when condition reset
   // ...
end
```

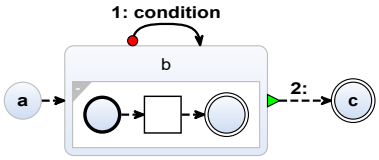


Fig. 10: Blech when-reset statement and synthesized SCChart.

The cobegin translation is illustrated in Fig. 7. As already mentioned in Sec. I-A, weak branches are handled by final regions.

G. If-else

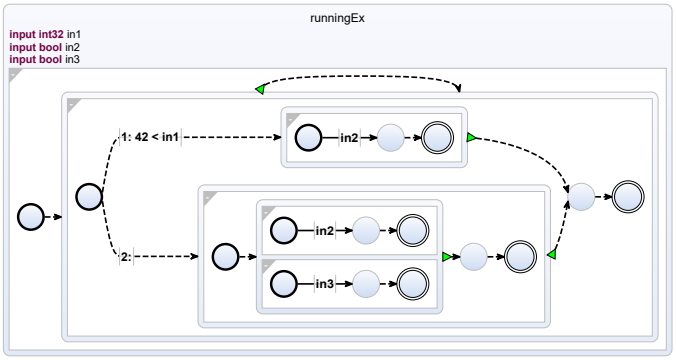
The if-else transformation is rather straightforward and illustrated in Fig. 8. If there are more else cases, more super states are added accordingly.

H. When-abort

Fig. 9 illustrates the translation of the when-abort statement. The inner behavior is aborted when a specific condition is met, but only after the execution in the body was halted for the first time. After that, the condition is checked at the beginning of each reaction.

```
activity runningEx
   (in1: int32, in2: bool, in3: bool)
   repeat
      if in1 > 42 then
         await in2
      else then
         cobegin
            await in2
         with
            await in3
         end
      end
   end
end
```

(a) Blech code.



(b) Mode diagram synthesized by Phase 1.

Fig. 11: Running example to illustrate Phase 1 and, in Fig. 16, Phase 3.

I. When-reset

The when-reset is similar to the when-abort, except that the inner behavior is restarted when the condition is met. This is illustrated in Fig. 10.

J. Running example after the translation step

The code in Fig. 11a will serve as a running example to illustrate the different phases. Fig. 11b shows the visualization of the running example after the initial translation step. The single statements have been transformed into their set SCChart counterparts. The bodies of hierarchical constructs are contained inside complex states. The activity name and the parameters were preserved. Await statements were realized with delayed transitions.

III. Phase 2: Label Extraction

As could be seen in Fig. 11b, the mode diagram synthesis described so far produces states with empty labels (except for activity call states). In order to talk about individual states in the figures we had to additionally inscribe labels such as “a” or “b”. This shows the necessity to provide the user with a mechanism to specify labels for code locations which are then transferred to the corresponding states in SCCharts. In this way visualized states may have meaningful names that can be referred to in a code review or documentation.

```
@@[label="aLabel"]
await condition
```

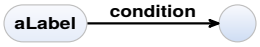


Fig. 12: Blech await statement with label specification and synthesized SCChart.

We propose to use Blech pragmas for the specification of labels. Pragmas are distinguished by a double @ prefix. Essentially, they act like comments with a special meaning in the source code. We adopt a @@[keyword="text"] syntax wherein various keywords denote the various labeling options as explained in the following.

We discuss two approaches to labeling the code locations: a state-only and an extended approach.

A. State-only labeling

The await statement is the primary source of states and transitions in our diagrams. Often, the code before the await implements one mode of operation and the code after the await implements the next mode. For instance, in our introductory example code in Fig. 1a, the code before line 10 is concerned with the initialization, while the code after line 10 starts a measurement. Here we would like to give a meaningful name, such as *init*, to the state the program is in while waiting for a certain condition. In Fig. 1a a simple comment is used to describe this location. To reflect this in the diagram, we propose a @@[label="text"] pragma which is placed *before* an await statement to label the state that the awaiting transition emanates from, as illustrated in Fig. 12.

B. Extended labeling approach

The *extended approach* allows for a more fine grained labeling of the elements. As we will see later, after some simplification steps, the only hierarchies that remain in the SCChart are due to activity calls or cobegin constructs. Since activity calls already have a label given by the activity they are referencing, no extra label is needed. Hence we consider the visual elements representing a cobegin construct. The elements to be labeled are different regions, each representing a cobegin branch, and the complex node containing these regions. For our extended labeling approach, we add the cobegin and branch pragma keywords.

As shown in Fig. 13, the labels are assigned using @@[cobegin="text"] for the complex node and @@[branch="text"] for a branch. All labels specified in the Blech code for the visualization must be placed before the cobegin construct. Consequently, a branch label is needed for each branch, when labeling the different regions, in order to make the labeling unambiguous.

IV. Phase 3: Transient State Elimination & Hierarchy Flattening

The translation above produces rather bloated diagrams as can be seen in our running example in Fig. 11b. For instance, the if and repeat statements induce unnecessary hierarchies,

```
@@[cobegin="calculation label"]
@@[branch= "calculation A"]
@@[branch= "calculation B"]
cobegin
  // ...
with
  // ...
end
```

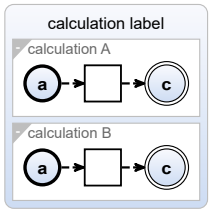
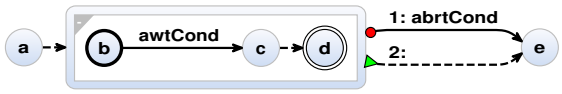
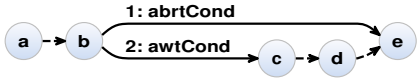


Fig. 13: Region labeling with the advanced labeling approach.



(a) Before flattening.



(b) After flattening.

Fig. 14: An example of how hierarchy is flattened for the when-abort construct.

which only obscure the stateful nature of the code. Furthermore, some of the generated states turn out to be superfluous, too. In our example, in every super-state there is a final state that is entered and left immediately. Such transient states do not convey any information and will be removed as well.

Thus after generating a diagram in Phases 1 and 2, we propose to simplify the result in Phase 3, which is divided into two steps. Step 1 flattens hierarchy levels, as discussed in Sec. IV-A. In Step 2 we collapse superfluous immediate transitions, as covered in Sec. IV-B. Step 2 follows after Step 1 because hierarchy flattening may produce additional immediate transitions that can be collapsed as well.

A. Hierarchy Flattening

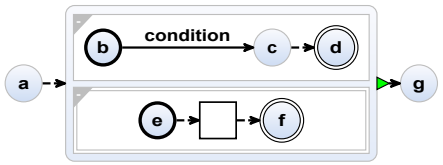
Most translation rules in Sec. II introduce complex states with inner behavior. Those states add hierarchy to the mode diagram, which may or may not be desired by the modeler.

1) Aborts: Fig. 14a shows the mode diagram produced by Phase 1 for the following snippet of code.

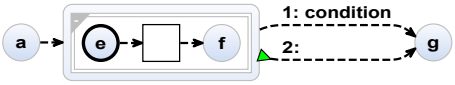
```
when abrtCond abort
  await awtCond
end
```

The hierarchical structure of the code is reflected in the hierarchical construction of the diagram. Note that the only state with a delayed outgoing transition is state b. Thus the only way that the super-state comprising b, c and d can be aborted is if it is waiting in b and *abrtCond* becomes true. This insight is explicitly shown in Fig. 14b. The hierarchy is removed (or, as we say here, *flattened*), all states are preserved, only their final or initial status is changed to a regular status, and the aborting transition directly links states b and e.

In this example, one may argue that removing the hierarchy has simplified the diagram. However, we have also lost some of the structure of the original Blech code. Furthermore, if



(a) Before flattening.



(b) After flattening.

Fig. 15: The weak-abort pattern.

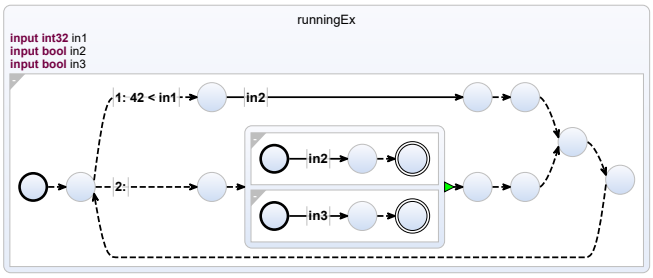
the body that is aborted contains not just one but many states, we would have to add an abort-transition for each of these states, which goes agains the "write-things-once principle" that generally motivates hierarchy in Statecharts. Hence, we propose to allow the user to configure whether hierarchy for aborts should be eliminated or not. As confirmed by the evaluation in Sec. V, users do appreciate this flexibility.

2) Activities: A run statement is represented by a special hierarchical state that *calls* into the respective sub-chart. Flattening this call essentially means to inline the mode diagram of the sub-chart in place of the call state. If done recursively from the entry point diagram, the result would be a global mode diagram of the entire program. However, for non-trivial programs this may become confusing. Activities represent self-contained parts of the source code and thus we generally recommend to also inspect their respective mode diagrams separately. Sometimes, however, code might be split into sub-activities for some implementation specific reasons but logically they are best understood together. For such cases we propose to make the inlining of call hierarchies configurable in the visualization tool.

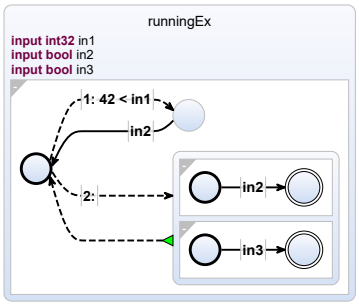
3) Cobegin & Weak-Abort pattern: Recall that the cobegin construct represents concurrency, and that the synthesized mode diagram has multiple concurrent regions. In general we cannot flatten these regions without constructing a cross-product of the region's states. It would be counter-intuitive to clutter the visualization with this artificial complexity. In fact, the very purpose of cobegin is to disassemble a complex behavior into separate easy-to-grasp behaviors. However, we can identify special patterns where a simplified visualization is possible.

A common pattern is to use cobegin to model weak preemption, where one branch contains only a single await statement (the preemption condition) and the other branches are weak and have some stateful behavior inside. The weak branches will be weakly aborted when the branch containing only the await statement terminates. Fig. 15 illustrates the flattening with the weak-abort pattern, with a strong awaiting branch.

Our implementation can also flatten hierarchies introduced by the other Blech statements, such as conditionals, but for brevity we skip their discussion.



(a) Result after hierarchy flattening.



(b) Result after eliminating transient states.

Fig. 16: Effect of Phase 3 on the running example diagram from Fig. 11b

Note that flattening hierarchy and removing complex states has implications on the labels described in Sec. III. Labelled complex states are due to either run statements or cobegin constructs. Their labels will be discarded when flattening the hierarchy. Labels which are attached to simple states, as discussed in Sec. III-A, are not affected by the flattening step.

4) Running example after flattening of hierarchy: Fig. 16a shows the running example after the hierarchy flattening step. The hierarchies of the if–then–else and repeat statements are flattened. Incoming edges changed their targets to the former initial states, whereas outgoing edges changed their source to the former final states. The hierarchy of the cobegin construct was not flattened, since it does not match the described weak-abort pattern. However, there are many transient states, some of them have been introduced by the flattening step. This motivates the optimization described next.

B. Transient State Elimination

Transient states are states whose outgoing transitions are all immediate. These states serve as connection points during the translation and flattening phases, but they do not represent any mode of operation of the program. In order to obtain a concise mode diagram we should remove transient states as far as possible. Transient states are removed by connecting each predecessor state with each successor state by a direct transition, possibly merging the guards, if there are any.

The challenge is to maintain the correct use of immediate termination transitions. They are used for exiting complex states as discussed earlier. Some of them are transformed during the flattening of hierarchies as covered in Sec. IV-A, but

not all. Plus, depending on configuration settings specifying that some degree of flattening is not desired, more termination transitions are present. Their presence has to be considered when proposing the collapse of transitions.

Fig. 16b shows the final mode diagram for the running example after eliminating transient states. This is now quite compact and we argue that its structure is probably close to how a human programmer would think about the Blech code. For technical details on how this translation is achieved, under consideration of preserving labels as well as possible, we refer to [8]

V. Evaluation

To validate the approach proposed here, we have implemented a prototype for extracting mode diagrams from Blech code and have asked four Blech developers from industry to evaluate it. The developers were given visualizations of real Blech code examples as well as artificial examples that illustrate the different layout options as described earlier. The experts were asked to validate the given visualizations for correctness, to assess whether the visualization is helpful to understand the given code, and to give their opinion on the representation. For those examples with several visualization options, they were asked to chose the one they liked most and explain their choice.

The general feedback was that the visualizations are helpful, especially if the code base is unknown or things are discussed with people who are not familiar with the code base. According to their overall assessment, the generated mode diagrams offer good insights, but still leave room for further improvement.

One finding was that it is hierarchy flattening is desirable in most but not all cases. Generally, hierarchies induced by run and abort statements should be preserved. Therefore, having flattening configurable is desired and will be kept.

The feedback to the proposed concept and suggestions by the experts revealed that labeling is a difficult topic, where experts have very individual preferences. Generally, there are multiple options to realize labels in terms of their declaration, place and handling (merging, discarding, etc.). A quantitative validation of different variations with a broad group of developers should help to determine the best solution for default settings.

One observation was that the indication of weak branches of cobegin constructs is too subtle. This is plausible, as the added rectangle for final regions in SCCharts is a light grey color on a white background. A straightforward approach to remedy this would be to simply use a stronger color to indicate final regions or including the "weak" keyword in the label. However, both solutions may be rather confusing and are not a distinct visual clue.

Finally, it became clear that knowledge of SCCharts is required to fully understand the visualization. Some form of learning or guide for users should be available for the tool to do its job without requiring the user to do research on their own. The need to understand SCCharts semantics raises the question of course, whether such details are needed or if a basic state chart with the most basic transitions would suffice, as further discussed in the conclusions in the last section.

VI. Related Work

For synchronous languages, there has been previous work to transform imperative programs into statechart representations. For example, Prochnow et al. proposed to synthesize SyncCharts [9] from Esterel. While our approach follows the same principle of structural statement-by-statement translation, the general goal differs. Previous approaches focused on full semantic equivalence between the programming languages and the graphical modeling notation. We here aim to provide automatically generated documentation with a higher level of abstraction. Hence, we focus on control-flow structures that expose the implicit stateful nature of the underlying program, and explicitly leave out most data-related execution aspects.

Outside the context of synchronous languages, Sen and Mall [13] present ways to extract state machines from object-oriented languages. They analyze the behavior of classes and infer state machines describing the class behavior. Approaches by Giomi [5] and Said et al. [11] describe state machine extractions based on control flow to represent the program's state space. Giomi processes hardware description languages and creates states based on wait statements in the control-flow graph. Said et al. propose a similar approach for C code but apply special state variable mining to detect relevant states. Compared to these approaches, we do not aim to visualize the implicit state space of a Blech program but its inherent operation modes, explicitly expressed in await statements.

To recover implicit state machine structures in imperative code, Somé and Lethbridge [15] use a pattern-based approach. They detect special nested choice patterns and switch statements and interpret and visualize these as state machines. This procedure is not necessary for a synchronous language such as Blech, since the await already explicitly separates states.

Another direction in this context, especially when working with legacy C code, is to generate a graphical representation for the program to compensate missing documentation and improve understandability. Smyth et al. [14] transform control structures and statements of a C programs into equivalent SCCharts. The EHANDBOOK[1] allows to extract graphical models from C code. It applies a more dataflow-oriented approach based on a flat program dependence graphs. In contrast to these approaches, we here focus on a more abstract view of the underlying program and its modes of operation, rather than a fully functional graphical replacement.

Aside from language-specific solutions, there is also the general field of software visualization [6]. The goal is to improve the understanding of given software artifacts by using various different graphical representations. One example in this area is the ExplorViz [4] tool. It takes an entire software system as input and creates an interactive "3D city" model to display the communications between the different components,

[1] http:/www.etas.com/ehandbook

packages, and classes. Rentz et al. [10] present an interactive graphical approach for exploring bundles, their dependencies, and services in OSGi software projects. Our Blech mode diagrams likewise emphasize a more high-level view on the software but may also be used during development to have an on-the-fly graphical representation of a specific activity. This concept of *transient views* is similar to tools like mbedder[2] that feature fluent and interchangeable editing of a textual and graphical representations.

VII. Conclusions and Outlook

We have presented an approach to automatically extract mode diagrams from the so far purely textual Blech language. The stateful nature of Blech allowed for a natural mapping from Blech code to modes/states and transitions. We postulate that such a translation would also be natural for other textual languages for developing reactive systems.

The mode diagrams are effectively a slightly extended and abstracted variant of SCCharts. This choice made it simple to capture precisely the semantics of the state-relevant part of the Blech language, which is not surprising given that Blech was inspired by the sequentially-constructive model of computation realized in SCCharts [7]. Furthermore, we could harness the existing open-source infrastructure for SCCharts to create the mode diagrams, crucially including the ability to automatically compute a layout for them.

The preliminary feedback from industrial users was encouraging, but also pointed to a number of still open questions and issues. The users also confirmed that preferences on mode diagrams are quite individual. Thus, flexibility and configurability for the mode diagram synthesis seems key. This may also include, as part of possible future work, the ability to choose how much "semantic accuracy" the mode diagrams should provide. Our focus so far was to provide a diagram synthesis that abstracted from run-time behavior in particular concerning data handling, but that still precisely captured possible state traces in reactions. That goal implied the need to differentiate, e. g., between immediate and delayed transitions and between different types of abort. However, at least for some users that was still too much detail, and made the mode diagrams less obvious to understand than a plain state diagram with just one type of transition would have been.

Another possible improvement, also according to the feedback from the industrial users, would be the merge of awaiting transitions followed by a decision point (i. e. if-else, repeat-until). That merge is not trivial regarding the triggers on the transitions. If one would simply merge the triggers from the delayed transition with the branches of the if-else, the label of the await would be duplicated. It is doubtful that this is an optimal and compact solution. Further problems arise if variables are used in the if-else conditions and the await statement. Complicated conditions might emerge this way. Boolean optimizations and smart simplifications have to be made here to make the merge viable.

[2]http://mbeddr.com/

On a practical implementation side, a full integration into Visual Studio Code would allow for example to link the elements in the code with their visual representation, as is already the case in KIELER when working with textual SCCharts and their visualization. Clicking in the visualization could highlight the corresponding parts in the code and vice versa, a feature that several industrial users had asked for.

References

[1] Charles André. Computing SyncCharts reactions. *Electr. Notes Theor. Comput. Sci.*, 88:3–19, 2004.

[2] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, January 2003. IEEE.

[3] Gérard Berry. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.

[4] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT'13)*, pages 1–4, September 2013.

[5] Jean-Charles Giomi. Finite state machine extraction from hardware description languages. In *Proceedings of Eighth International Application Specific Integrated Circuits Conference*, pages 353–357. IEEE, 1995.

[6] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, 2005.

[7] Friedrich Gretz and Franz-Josef Grosch. Blech, imperative synchronous programming! In *Proc. Forum on Specification Design Languages (FDL' 18)*, pages 5–16, September 2018.

[8] Daniel Lucas. Extraction of mode diagrams from Blech. Master thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, April 2020. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/dalu-mt.pdf.

[9] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '06)*, Ottawa, Canada, June 2006.

[10] Niklas Rentz, Christian Dams, and Reinhard von Hanxleden. Interactive visualization for OSGi-based projects. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 84–88, Adelaide, Australia, September 2020. IEEE.

[11] Wasim Said, Jochen Quante, and Rainer Koschke. On state machine mining from embedded control software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 138–148. IEEE, 2018.

[12] Francisco Sant'Anna, Roberto Ierusalimschy, Noemi de La Rocque Rodriguez, Silvana Rossetto, and Adriano Branco. The design and implementation of the synchronous language céu. *ACM Trans. Embedded Comput. Syst.*, 16(4):98:1–98:26, July 2017.

[13] Tamal Sen and Rajib Mall. Extracting finite state representation of Java programs. *Software & Systems Modeling*, 15(2):497–511, 2016.

[14] Steven Smyth, Stephan Lenga, and Reinhard von Hanxleden. Model extraction for legacy C programs with SCCharts. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '16), Doctoral Symposium*, volume 74 of *Electronic Communications of the EASST*, Corfu, Greece, October 2016. With accompanying poster.

[15] Stéphane S. Somé and Timothy C. Lethbridge. Enhancing program comprehension with recovered state models. In *Proceedings 10th International Workshop on Program Comprehension*, pages 85–93. IEEE, 2002.

[16] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.