Chapter 10
SCXML on Resource Constrained Devices

Stefan Radomski, Jens Heuschkel, Dirk Schnelle-Walka, and Max Mühlhäuser

Abstract Ever since their introduction as a visual formalism by Harel et al. in 1987, state-charts played an important role to formally specify the behavior of reactive systems. However, various shortcomings in their original formalization lead to a plethora of formal semantics for their interpretation in the subsequent years. In 2005, the W3C Voice Browser Working Group started an attempt to specify SCXML as an XML dialect and corresponding semantic for state-charts and their interpretation, promoted to W3C recommendation status in 2015. In the context of multimodal interaction, SCXML derives a special relevance as the markup language proposed to express dialog models as descriptions of interaction in the multimodal dialog system specified by the W3C Multimodal Interaction Working Group. However, corresponding SCXML interpreters are oftentimes embedded in elaborate host environments, are very simplified or require significant resources when interpreted. In this chapter, we present a more compact, equivalent representation for SCXML documents as native data structures with a respective syntactical transformation and their interpretation by an implementation in ANSI C. We discuss the characteristics of the approach in terms of binary size, memory requirements, and processing speed. This will, ultimately, enable us to gain the insights to transform SCXML state-charts for embedded systems with very limited processing capabilities and even integrated circuits.

S. Radomski (✉)
TU Darmstadt, Telekooperation Group, Darmstadt, Germany
e-mail: radomski@tk.tu-darmstadt.de

J. Heuschkel
TU Darmstadt, Telekooperation Group, Darmstadt, Germany
e-mail: heuschkel@tk.tu-darmstadt.de

D. Schnelle-Walka
S1nn GmbH & Co. KG, Stuttgart, Germany
e-mail: dirk.schnelle-walka@s1nn.de

M. Mühlhäuser
TU Darmstadt, Telekooperation Group, Darmstadt, Germany
e-mail: max@tk.tu-darmstadt.de

D.A. Dahl (ed.), *Multimodal Interaction with W3C Standards*,
DOI 10.1007/978-3-319-42816-1_10

10.1 Introduction

The State-Chart eXtensible Markup Language (SCXML) is a W3C recommendation for a specific syntax and semantics of Harel state-charts [4] as a compact visual formalism for state-transitioning systems. It was finalized in September of 2015 [6] and is suggested in the W3C Multimodal Architecture and Interfaces recommendation [1] as a possible description of interaction managers to control modality components in a multimodal user interface. While state-charts, as a visual formalism, were already proposed by Harel in 1987, deficiencies with the initial semantic [5] lead to the development and scientific publication of more than 40 different semantics in the subsequent years [3, 9].

As such, the endeavor of SCXML to standardize the syntax and semantic via the W3C is direly needed to reestablish compatibility of the various tools and platforms available to model and interpret Harel state-charts. However, the syntactical description of SCXML as an XML dialect and several language features implied by tests in the SCXML Implementation Report Plan (IRP) strongly suggest an implementation via interpretation at runtime with a full XML document object model still available. While this overall approach has been spectacularly successful, e.g., with HTML [2] and enables considerable flexibility to dynamically adapt the XML description via scripting during interpretation, it severely limits the applicability of SCXML to platforms with sufficient computing power and memory.

In the following sections, we will describe an approach to preprocess SCXML documents into more suitable data structures and present an implementation of the `microstep(T)` function in ANSI C. This implementation, by a large margin, outperforms the pseudo-code description of the same algorithm in Appendix D of the SCXML recommendation. This is relevant as many SCXML interpreters do indeed align their implementation of this central piece of functionality with the pseudo-code description. Furthermore, by employing the syntax and semantics of ANSI C as a formal programming language, we do address one point of critique with the pseudo-code in Appendix D, that is to provide an actually executable description for `microstep(T)`.

While the evaluation of the ANSI C algorithm will already show general applicability for even the smallest off-the-shelf micro-controllers, the last part of this chapter will describe a first approach for a transformation from SCXML onto VHDL as a hardware description language. Such a description would allow to mold SCXML documents into Field Programmable Gate Arrays (FPGAs) and even Application Specific Integrated Circuits (ASICs), which we expect to gain elevated relevance in the scope of applications for the Internet Of Things (IoT).

10.2 Semantic of SCXML

Before we dive into the actual algorithms, we will need to define some important sets and relations from the SCXML recommendation that are relevant to retrace the algorithms' functionality and convince ourselves of their correctness. This section does assume a passing familiarity with the SCXML recommendation or, at least, with Harel state-charts in general.

The interpretation and execution of an SCXML document at runtime can be conceived as a series of *microsteps* over a set of transitions (T) enabled by an event e. At any point in time, the interpreter is in a given *configuration* as a set of proper states that are said to be *active*. Any change to the configuration of an interpreter is assumed to be instantaneous (perfect synchronicity hypothesis [9]) and always caused by events that enable transitions. A special non-event ε is introduced in the SCXML recommendation to extend this notion for spontaneous transitions. Figure 10.1 summarizes the sequence of activities for an interpreter within a `microstep(T)` iteration. Every iteration starts with establishing the current event as follows:

1. If the previous iteration did not exhaust spontaneous transitions, set the current event to ε as the non-event (**1a**) that only enables event-less (spontaneous) transitions.
2. If there were no more spontaneous transitions enabled by ε in the previous iteration, dequeue an event from the *internal* event queue (**1b**).
3. If there are no events remaining on the internal event queue, attempt to dequeue from the *external* queue (**1c**) or block execution until an event becomes available. After a series of such micro-steps and before dequeuing an event, the interpreter is said to have performed a *macro-step* and reached a new stable configuration. At this point, a compliant interpreter has to make sure that all invocations for external systems specified via `<invoke>` within states of the active configurations are started and all other such invocations stopped.

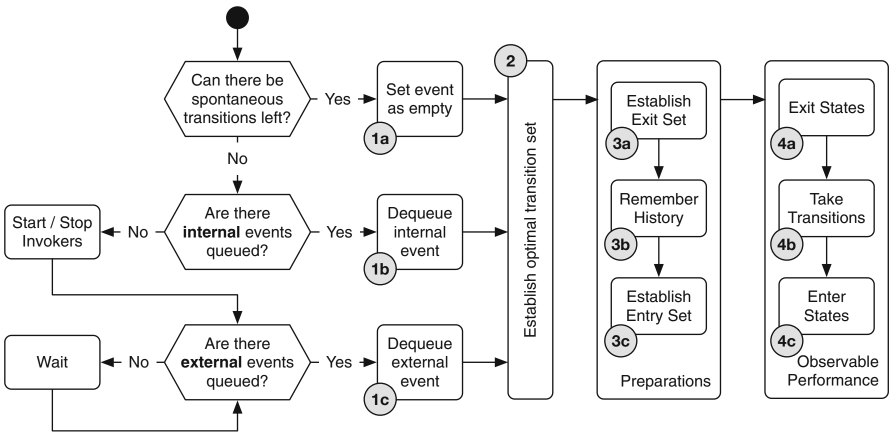

Fig. 10.1 Flow of activities within a micro-step iteration

Now, whenever we are about to proceed to the next activity (**2**), we can assume an event to be set (be it ε or an actual event). For this given event e, we will have to establish the *optimal transition set*. To this effect, the SCXML recommendation defines a containment hierarchy of transition sets as follows:

- **Active Transitions**
  All `<transition>` elements contained as direct children of states in the active configurations are said to be *active*. They form the superset of all other transition sets below.
- **Matched Transitions**
  The subset of active transitions, with at least one event descriptor in their `event` attribute matching the current event's name are said to be *matching*. If the current event is the ε event, all active transitions with no `event` attribute (spontaneous transitions) are matched.

  It is allowed for an event descriptor to have a `.*` suffix for compatibility with CCXML. Furthermore, it is legal for a transition to specify multiple, space-separated event descriptors in their `event` attribute. In this case, an event matches a transition if one of the transition's event descriptors matches the event's name.
- **Enabled Transitions**
  The set of matching transitions is further reduced by requiring an eventual `cond` attribute to evaluate to `true` on the *data-model* (usually an embedded scripting language context). A matched transition with a condition that holds or without a condition is said to be *enabled*.
- **Optimally Enabled Transitions**
  For a transition to be optimally enabled, there can be no earlier enabled transition with the same source state, neither can a transition in a descendant state of our source be enabled. The first criterion provides an ordering for enabled transitions within the same state. The second criterion allows to *specialize* a state-chart's behavior in response to events by overriding behavior in a more deeply nested sub-state of a composite state.
- **Optimal Transition Set**
  Generally, it is not possible for all optimally enabled transitions to be taken in the same micro-step as they might lead to an invalid subsequent configuration. Therefore, the optimal transition set is established as the largest set of non-conflicting, optimally enabled transitions. Here, two transitions are said to be conflicting, if the intersection of their exit sets is nonempty. For any two such transitions, the one with the highest *priority* will be added to the optimal transition set. The priority of a transition is defined very similar to the precedence with the optimally enabled transition set in such that a transition t_1 has a higher priority than t_2 if (1) the source of t_1 is a descendant of the source of t_2, (2) or t_1 precedes t_2 in document order.

The optimal transition set at the end of the above containment hierarchy now contains all the transitions T that are to be performed in response to an event in the current `microstep(T)` iteration. It is crucial for any performant implementation of SCXML to be able to identify this optimal transition set efficiently as it is calculated at least twice for any non-ε event: once as the set of transitions to be taken for the event itself and, subsequently, at least once for the ε event to exhaust any spontaneous optimal transitions in the new configuration.

We will see later that the transitions in the optimal transition set already define the microstep's *exit-set* (**3a**) as the set of active states to be exited within the current micro-step. For any composite state in this set that contains a `<history>` pseudo state as a child we will have to remember its active children or, depending on the histories `type`, even all its active descendants (**3b**) to be reentered when the `<history>` pseudo state is in the target-set of a subsequent iteration.

The optimal transition set also already defines an intermediate *target-set* as the set of states directly referenced in the transitions' `target` attributes. From this target-set, we can establish the *entry-set* of the optimal transition set (**3c**) by calculating its *completion*, which defines the set of states to be actually entered within the current micro-step. The completion of a state in the target-set depends on its type and we will discuss all of them in more detail when we step through the actual `microstep(T)` algorithm below.

After we established the `exit-`, `transition-`, and `entry-set` for a given event in a state-chart's configuration, we can perform the actual micro-step as (**4a**) exiting states, (**4b**) transitioning and (**4c**) entering the new states. This will update the state-chart's active configuration and invoke any*executable content* associated with these activities.

10.2.1 Scope of the Algorithm

The `microstep(T)` function outlined above is at the core of every SCXML interpreter and its description constitutes the bulk of the SCXML recommendation. There are, however, additional responsibilities for a compliant interpreter that we do not address in the algorithm we are about to describe below:

- We do not concern ourselves with invocations of external components via the `<invoke>` element. Such invocations are to be processed prior to dequeueing an external event, right before the interpreter is said to have performed a macro-step. It is perfectly possible to trigger these invocations via our algorithm, but the transformation onto ANSI C we implemented will, currently, only process a single state-chart per file and virtually all tests defined for `<invoke>` in SCXML assume a nested state-chart to be processed.
- We do not support any I/O processor other than the SCXML I/O processor.
- We have not implemented file operations or any retrieval of content referenced via a URL.

We do, however, support the transformation of executable content into semantically equivalent control flow constructs in ANSI C for various callbacks into user-supplied code as well as various datamodel implementations. Both features are required to pass any meaningful subset of the SCXML IRP tests and evaluate the algorithm. The datamodel integration is not part of the actual algorithm but assumed to be available as a set of respective callback functions that will evaluate the various expressions.

10.3 Preparing SCXML Data Structures

If we are to target embedded platforms, it seems wasteful not to preprocess the SCXML documents into a more compact representation. While there are XML parsers available that compile into binary code as small as 30 KB,[1] they only offer a streaming API for XML documents and still require us to establish and maintain a suitable representation at runtime. As such, we might as well preprocess the SCXML documents into a native representation and pre-calculate several sets and relations that will become relevant when we discuss the actual `microstep(T)` algorithm below.

10.3.1 States

When we regard the states of an SCXML document, we can encode all the information required for a semantically equivalent execution of a given state-chart via the compound data structure given in Listing 10.1. During transformation, an array of such structures is defined, containing all the states (along with the pseudo-states and the root state) of an SCXML state-chart. The states in this array are sorted by document-order, which corresponds to entry-order and reverse exit-order for the `microstep(T)` algorithm.

Listing 10.1 Representing a state as a compound data structure

```
struct state {
  const char*            name;
  const uint8_t          type;
  const uint16_t         parent;
  const exec_content_t   on_entry;
  const exec_content_t   on_exit;
  const char             children[STATE_BYTES];
  const char             completion[STATE_BYTES];
  const char             ancestors[STATE_BYTES];
  const elem_data*       data;
};
```

[1] https://dev.yorhel.nl/yxml.

- The **`name`** field contains the eventual identifier of the state or is `NULL` if the state does not specify an identifier. This identifier is only needed for the `In ('state')` predicate and is not used during the actual `microstep(T)` algorithm below.
- The **`type`** field identifies the states type in the original SCXML document and can be one of `{PARALLEL, COMPOUND, ATOMIC, FINAL, INITIAL, HIST_DEEP, HIST_SHALLOW}`. The most significant bit is reserved for the `HAS_HIST` flag, which denotes (1) whether there is a `<history>` child element for a composite state or (2) whether there is another `<history>` element in the descendants of a given history's parent state. This flag will become important later when we complete a history element in the target set onto its entry set.
- The **`parent`** field identifies the index of this state's parent in the array of all states per document.
- The **`on_entry`** and **`on_exit`** fields are pointers to static functions where the respective executable content is generated.
- The next three fields, **`children`**, **`completion`**, and **`ancestors`** are bit arrays with a width sufficient to model every state from the original SCXML document as a single bit. The `children` and `ancestors` bit arrays are initialized such that the bit at index N is set if the state at index N is in the respective relation to the current state. The semantic of the `completion` bit array is more ambiguous and depends on the state's type:
  - For <**`parallel`**> states, it identifies all the direct, proper child states.
  - For **compound** `<state>`s, the completion identifies the first child in document order or the states from the target set identified by the state's `initial` attribute.
  - For <**`final`**> and **atomic** `<state>`s, the completion is empty.
  - For <**`initial`**> pseudo states, it identifies the states in the target set of a contained `<transition>` element.
  - For <**`history`**> pseudo states, its semantic is rather complicated. Essentially, it identifies all the parent's descendant states that are *covered* by the history, i.e. the parent's proper child states for `shallow` histories. For `deep` histories, however, it does not necessarily identify all proper descendant states, but only those that are not already covered by a nested `<history>` pseudo state. We will see later that this construction allows us to model all of the state-chart's history as a single bit-array with a width corresponding to the number of states only.
- Finally, the **`data`** element contains a pointer to an NULL terminated array of compound data structures, representing the optional `<data>` elements for late initialization upon first activation with a late data binding. For an early data-binding, all these `<data>` elements are attached to the state-chart's `<scxml>` root state.

The memory layout of an individual compound data structure for a state is depicted in Fig. 10.2 with its actual size depending on the target architectures bit-width and the total number of states in a state-chart. Figure 10.3 shows the size of a single state structure as a function of the total number of states when assuming a 16-bit target architecture. It is noteworthy that the three bit-arrays (`children`, `completion`, and `ancestors`) are the largest contributors to its size and will completely dominate the required memory for large number of states.

Figure 10.3 also allows to determine the total amount of memory required to represent all of a state-chart's states as an array of these compound data structures by counting the number of all states in an SCXML document and multiplying it with the function value at the given point (e.g., to encode 100 states, we would need round about 100×60 bytes). This is in addition to the memory required for the string literals for the states' identifiers pointed to by the `name` field. If we were required to reduce this memory, we have many options to trade runtime for memory in this data structure, e.g. to

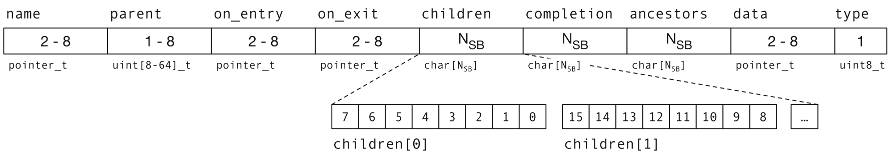

Fig. 10.2 The memory representation of a single state structure depends on the target platform's bit-width, the total number of states and any eventual padding

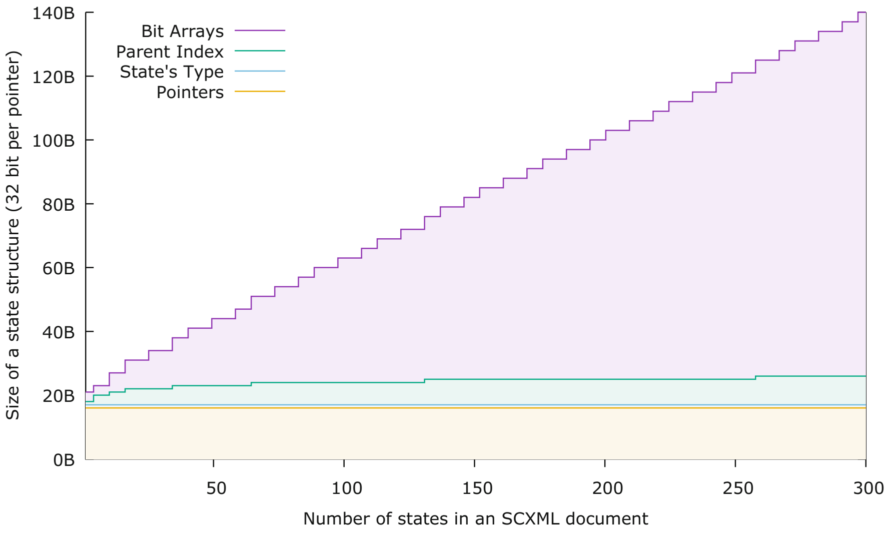

Fig. 10.3 Aggregated size of a single state structure as a function of the total number of states in an SCXML document without alignment padding

- Calculate the `children` via the `parent` relation.
- Calculate the `ancestors` via the reversed `children` relation.
- Calculate the `completion` via the state's `type` along with its `children` relation.
- Calculate the `parent` as the most significant bit in the `ancestors` relation.

However, some of these calculations can be quite expensive (most notably the completion of deep `<history>` states with nested `<history>` elements).

10.3.2 Transitions

Similar to the states above, we can establish an array of structures containing all the relevant information from the `<transition>` elements in the original SCXML state-chart. These compound data structures (Listing 10.2) will also already contain several pre-calculated, important sets and relations that are static with regard to a given SCXML state-chart and relevant for the execution of `microstep(T)`. The array is sorted in post-order traversal of all transitions and we will see later why this is very beneficial.

Listing 10.2 Representing a transition as a compound data structure

```
struct transition {
const uint16_t       source;
const char           target[STATE_BYTES];
const char           exit_set[STATE_BYTES];
const char           conflicts[TRANS_BYTES];
const uint8_t        type;
const char*          event;
const char*          condition;
const exec_content_t on_transition;
};
```

- The **`source`** of a transition identifies its parent state (proper, or otherwise) by the state's index in the array of all states.
- The **`target`** field is a bit-array in which a given bit is set, if the `<transition>` element identified the respective state in its `target` attribute.
- The **`exit_set`** field is a bit-array identifying the transition's *complete* exit set. The definition of the actual exit-set from the SCXML standard is as follows:

  The exit set of a transition in configuration C is the set of states that are exited when the transition is taken when the state machine is in C. If the transition does not contain a `target`, its exit set is empty. Otherwise (i.e., if the transition contains a `target`), if its `type` is `external`, its exit set consists of all active states in C that are proper descendants of the Least Common Compound Ancestor (LCCA) of the source and target states. Otherwise, if the transition has `type internal`, its source state is a compound state, and all its target states are proper descendants of its source state, the exit set consists of all active states in C that are proper descendants of its source state.

Unfortunately, the exit-set depends on the state-chart's configuration C. The implied assumption is that we can calculate the exit-set for the complete configuration in which every state is active and establish the actual exit-set at runtime by intersecting each transition's complete exit-set with the active configuration. We do not have a proof for this assumption, but it makes sense given the definition, the calculation in the pseudo-code from Appendix D in the SCXML recommendation and, indeed, all relevant IRP tests do pass.

- The **`conflicts`** field is a bit-array that identifies other transitions which can, for whatever reason, never occur in an optimal transition set with the given transition. If we look at the definition of the optimal transition set in the containment hierarchy from Sect. 10.2, we can syntactically identify several situations in which two transitions conflict:

  1. For two transitions to be *active* within the same iteration, their source states need to be active at the same time. This can only be the case if their least common ancestor is a `<parallel>` element.
  2. For two transitions to be *matched* at the same time, there has to be an event that matches both transitions. This can never be the case for event-less and eventful transitions or two eventful transitions that have no event descriptor that matches a common event.
  3. We cannot exploit any criteria with regard to the *enabled* transition set as we, usually, cannot make any assumption about the evaluation of an eventual `cond` attribute at transformation time.
  4. For two transitions to be *optimally enabled*, their source states cannot be identical or ancestrally related.
  5. For two transitions to be in the *optimal transition set*, their exit-sets may not overlap.

  This results in quite a number of sufficient criteria for two transitions to conflict and minimizes the amount of transitions to consider when establishing the optimal transition set per micro-step considerably.
- The **`type`** field is interpreted as a bit array that specifies the type of the transition, it might be one or any of `{SPONTANEOUS, TARGETLESS, INTERNAL, HISTORY, INITIAL}`. Not all of these are currently used in the actual algorithm below, though.
- The **`event`** field contains a pointer to the constant string literal with the transition's event descriptor list and is required to establish the *matched* transition set.
- The **`cond`** field contains a pointer to the constant string literal with the transition's condition and is required to establish the *enabled* transition set.
- Finally, the **`on_trans`** field is a pointer to a static function with the transition's executable content.

source	target	exit_set	conflicts	type	event	condition	on_transition
1 - 8	N_{SB}	N_{SB}	N_{TB}	1	2 - 8	2 - 8	2 - 8
uint[8-64]_t	char[N_{SB}]	char[N_{SB}]	char[N_{TB}]	uint8_t	pointer_t	pointer_t	pointer_t

Fig. 10.4 The memory representation of a single transition structure also depends on the target platform's bit-width, the total number of states and transitions, as well as any eventual padding

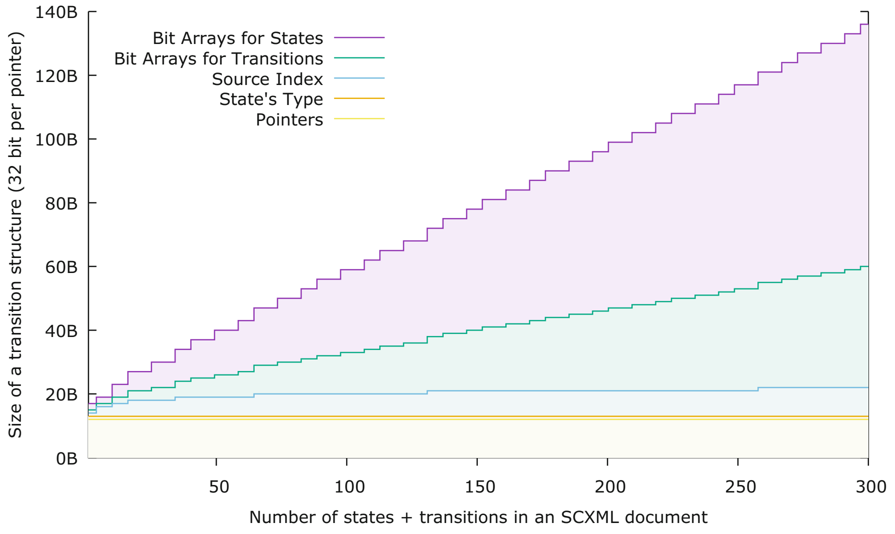

Fig. 10.5 Aggregated size of a single transition structure as a function of the total number of states plus transitions when assuming equal numbers

The memory layout of such a transition structure is given in Fig. 10.4 and, again, its size depends on the bit-width of the target platform and the total number of states and transitions in a given SCXML document. The size of this structure as a function of the total number of states and transitions, when assuming equal numbers and a 16-bit architecture is depicted in Fig. 10.5.

The relation of the structure's size with regard to the complexity of the complete document shows the exact same development as the one for states in Fig. 10.3, though, its increase in size is somewhat dampened if there are more transitions than states as we only need a single bit-array for transitions.

10.3.3 SCXML Context

The states and transitions above represent immutable, constant data for any given SCXML document and can be generated during transformation. But there is also a dynamic part for the interpretation of a state-chart, which we will represent as an SCXML context (see Listing 10.3). This allows us to maintain multiple instances of a state-chart at runtime as distinct contexts that share the states and transitions from above as static data.

Listing 10.3 The context of an SCXML instance at runtime

```
struct ctx {
    uint8_t  flags;
    char     config[STATE_BYTES];
    char     history[STATE_BYTES];
    char     initialized_data[STATE_BYTES];

    void* event;
    void* user_data;

    /* miscellaneous user supplied callback functions */
    dequeue_internal_t    dequeue_internal;
    dequeue_external_t    dequeue_external;
    is_enabled_t          is_enabled;
    is_true_t             is_true;
    raise_done_event_t    raise_done_event;

    /* user-supplied callback functions for executable content */
    exec_content_log_t          exec_content_log;
    exec_content_raise_t        exec_content_raise;
    exec_content_send_t         exec_content_send;
    exec_content_foreach_init_t exec_content_foreach_init;
    exec_content_foreach_next_t exec_content_foreach_next;
    exec_content_foreach_done_t exec_content_foreach_done;
    exec_content_assign_t       exec_content_assign;
    exec_content_init_t         exec_content_init;
    exec_content_cancel_t       exec_content_cancel;
    exec_content_script_t       exec_content_script;
};
```

- The **`flags`** field is a generic member of the context to remember various boolean values across invocations of a `microstep(T)`. Currently, it encodes (1) whether the state-chart's context is still in pristine condition and some setup is required (`CTX_PRISTINE`), (2) whether we already exhausted spontaneous transitions (`CTX_SPONTANEOUS`) and need to dequeue an event, and (3) whether the state-chart entered a top-level final state and is done (`CTX_TOP_LEVEL_FINAL`).
- The **`config`** field contains the state-chart's currently active configuration as a bit-array, such that the bit at index *i* is set if the corresponding state in the array of all states is active.
- The **`history`** field is another bit-array that encodes the valuation of all `<history>` elements (deep or shallow) from the original state-chart. It is not obvious how we can encode all of the history in a single bit-array and we will discuss this point in more detail below.
- The **`initialized_data`** field is a bit-array that encodes which states were already entered. This is only required for SCXML documents with a late data-binding and allows us to perform the initialization of eventual nested `<data>` elements only for the first activation of a state.

- The **event** field is an opaque pointer to a memory region containing the current event. The `microstep(T)` algorithm will indeed not know any details about the current event as we will just employ the user-supplied `is_enabled` callback to determine whether an event matched and enabled any transition under consideration.
- The **user_data** field is another opaque pointer, where user-supplied code can register any additional data that might be required per SCXML interpreter instance and has no purpose in the scope of the `microstep(T)` algorithm below.
- All other fields are callbacks into user-supplied code. Most notably:
  - The **dequeue_internal** and **dequeue_external** functions will return an opaque pointer for the current event.
  - The **is_enabled** callback is called with a transition structure and the opaque event pointer to determine whether the given transition is matched and enabled by the given event.
  - The **is_true** callback determines whether a given expression evaluates to `true` on the data-model.
  - The **raise_done_event** is called with a state structure, and the information from an optional <`donedata`> element to raise the respective `done.state.ID` event on the internal queue.

The other callbacks are invoked as part of the executable content in the various, generated `on_entry`, `on_exit`, and `on_trans` functions referenced from the respective function callbacks in the `state` and `transition` structures above.

The total size of the context structure as a function of the number of states in the SCXML document is given in Fig. 10.6. For documents with only a few states, the

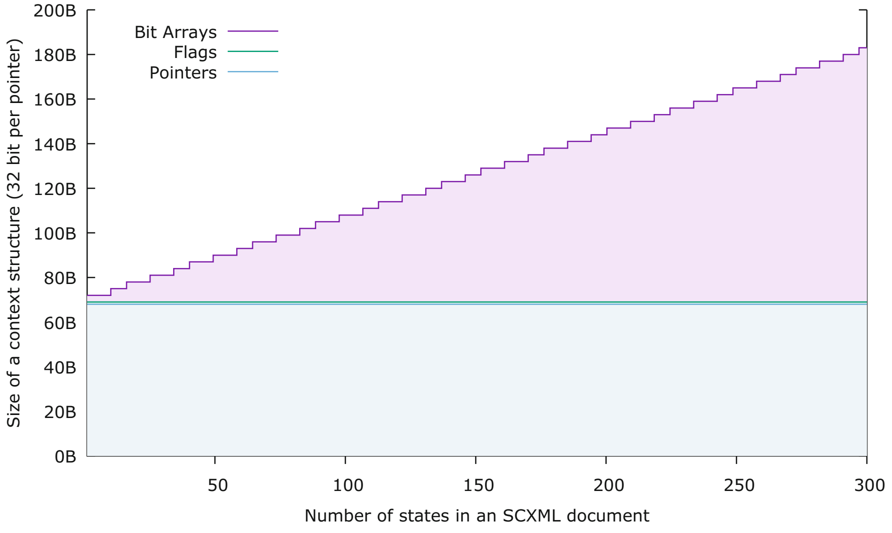

Fig. 10.6 Aggregated size of a single context structure as a function of the total number of states

size is dominated by the memory required to store the addresses of the callback functions for executable content and other datamodel queries. When the number of states increases, however, the size of the bit-arrays required to model the history, the active configuration and already initialized states start to become the major contribution. Ultimately, another such bit-array for current invocations per `<invoke>` would likely have to be added as well.

10.3.4 Other Elements

In order to enable the processing of executable content, we also need to encode various other SCXML elements into compound data structures. These merely encode the information contained within the respective elements and, as opposed to the states and transitions above, there are no special considerations with regard to their representation other than to make the information available to user-supplied code.

- The <**`data`**> elements, as children of the `<datamodel>` elements, are represented as compound data structures with four pointers to string literals for (1) their `id` attribute, (2) the `src` attribute, (3) the `location` attribute, and eventual content. If any of those attributes is unspecified, its value will be initialized as `NULL`.

  All compound data structures for `<data>` elements are contained within an array, with `NULL` entries in between as delimiters. To reference a consecutive set of data structures, the address of the first member is taken and a macro is provided to iterate all subsequent members until the next `NULL` delimiter.
- A similar approach is taken for all <**`param`**> elements. These are encoded in compound data structures, each with pointers to three string literals as (1) their `name` attribute, (2) the `location` attribute, and (3) their `expr` attribute. Again, to reference a consecutive set of such elements, the address of the first member is taken with macros to test for additional structures.
- The <**`donedata`**> elements are also just encoded as a compound data structure with (1) a `source` field as the index of the containing state, (2) their `content` attribute as a pointer to a string literal with the textual value of any contained `<content>` element, or (3) an eventual `contentexpr` attribute and (4) a pointer to the first `<data>` element in the array of all data structures.
- The <**`foreach`**>, elements within a document are encoded as simple compound data structures with three pointers to string literals for their attributes (1) `array`, (2) `index` and (3) `item`. Again, if an attribute is not specified with a `<foreach>` element, its value is `NULL`.
- Finally, <**`send`**> elements are encoded with all their possible attributes as compound data structures as well. Eventual `<content>` children are given as string literals in their textual representation and `<param>` elements are given as a reference to their first entry in the array of all `param` data structures.

10.3.5 Executable Content

As part of the transformation from SCXML onto C, we will also transform the executable content contained as children of the `<onexit>`, `<onentry>` and `<transition>` elements. This is not strictly required for an implementation of the `microstep(T)` algorithm, but it is easily done and extends the domain of the transformation.

To transform the executable content is to invoke the various callbacks in the SCXML context (cf. Listing 10.3) in the correct order and under the correct conditions. Furthermore, we will have to regard the error semantics of the various blocks. If an error occurs within a block of executable content, a compliant interpreter is required to raise a respective event and continue processing with the next block of executable content. To this effect, we encode every individual block of executable content as a static function that is exited if an error occurs and call each block sequentially from within a general `[ID]_on_entry`, `[ID]_on_exit`, `[ID]_on_trans` function. Here, the `ID` of an element is either the value of its eventual `id` attribute or a unique identifier derived from its position in the SCXML document.

With the callbacks given in the SCXML context and the representation of the various elements above, it is straightforward to see how we can generate C code to model the behavior of the executable content. The remaining SCXML elements of executable content for which we did not define a compound data structure above are merely passed via their various attributes into the user-supplied callbacks.

10.4 A Compact Algorithm for Interpretation

Now that we have all the data structures and control flow for executable content from an SCXML document defined, we can present the actual algorithm for `microstep(T)`. The algorithm is closely aligned with the sequence of activities from Fig. 10.1 and we will, indeed, describe its workings by presenting each activity in turn.

10.4.1 Preparations

Currently, the arrays of compound data structures for the transitions, states, and other SCXML elements are modeled as static global variables accessible throughout the compilation unit and their identifiers hard-coded into the algorithm. As such, there is very little preparation required but to allocate memory for a `ctx` and register the various callback functions:

Listing 10.4 Instantiating a state-machine context

```
int main(int argc, char** argv) {
   int err;
   ctx ctx;
   memset(&ctx, 0, sizeof(ctx));

   /* register callbacks */
   ctx.is_enabled = &is_enabled;
   ctx.is_true = &is_true;
   ...

   /* run interpreter until done */
   while(true) {
      err = microstep(&ctx);
      if (err == ERR_DONE)
        break;
      if (err != ERR_OK)
   return EXIT_FAILURE;
   }
   return EXIT_SUCCESS
}
```

The callbacks are not shown, but we did indeed implement them in order to pass the SCXML IRP tests.

10.4.2 Dequeuing Events

The first thing to do within the `microstep` function is to transition into the initial configuration if the state-machine is still pristine or to set the current event (Listing 10.5).

Listing 10.5 Initialization the state-chart and dequeing events

```
size_t i, j, k;
int err = ERR_OK;
char conflicts[TRANS_BIT_ARRAY]  = TRANS_BIT_ARRAY_INIT;
char target_set[STATE_BIT_ARRAY] = STATE_BIT_ARRAY_INIT;
char exit_set[STATE_BIT_ARRAY]   = STATE_BIT_ARRAY_INIT;
char trans_set[TRANS_BIT_ARRAY]  = TRANS_BIT_ARRAY_INIT;
char entry_set[STATE_BIT_ARRAY]  = STATE_BIT_ARRAY_INIT;
char tmp_states[STATE_BIT_ARRAY] = STATE_BIT_ARRAY_INIT;

if (ctx->flags & CTX_TOP_LEVEL_FINAL)
    return ERR_DONE;
if (ctx->flags == CTX_PRISTINE) {
    global_script(ctx, &states[0], NULL);
    bit_or(target_set, states[0].completion);
    ctx->flags |= CTX_SPONTANEOUS | CTX_INITIALIZED;
    goto ESTABLISH_ENTRY_SET;
}
```

```
20 if (ctx->flags & CTX_SPONTANEOUS) {
       ctx->event = NULL;
       goto SELECT_TRANSITIONS;
   }
   if ((ctx->event = ctx->dequeue_internal(ctx)) != NULL) {
25     goto SELECT_TRANSITIONS;
   }
   if ((ctx->event = ctx->dequeue_external(ctx)) != NULL) {
       goto SELECT_TRANSITIONS;
   }
```

Each iteration of a micro-step starts by allocating memory on the stack for all variables required in the function's scope as is required in ANSI C. Afterwards, we check to see whether the state-chart already entered a top-level final state (line 10–11), which signifies the end of all processing. If this is not given, we check whether the state-chart is still in pristine condition, in which case we execute any eventual global script elements and set the target set to the root state's completion as defined for compound states in Sect. 10.3.1 before we continue processing with the completion of the target set as the entry set (line 13–18). Otherwise we establish the current event (Fig. 10.1 **(1a–c)**) as `NULL` if spontaneous transitions were not yet exhausted per `ctx->flags` or attempt to dequeue an event and continue to establish the optimal transitions set (line 20–29).

10.4.3 Selecting Transitions and Establishing the Exit-Set

The next activity is to establish the optimal transition set (Fig. 10.1 **(2)**). This is a crucial section of the algorithm as it will be executed at least twice for any non-null event. The corresponding pseudo-code from Appendix D in the SCXML recommendation is rather obscure and very elaborate. For the ANSI C implementation in Listing 10.6, a single iteration of all transitions is sufficient with the majority of iterations skipped very early.

Listing 10.6 Establishing the optimal transition set

```
SELECT_TRANSITIONS:
for (i = 0; i < NUMBER_TRANSITIONS; i++) {
    if (transitions[i].type & (TRANS_HIST | TRANS_INITIAL))
        continue;

if (BIT_HAS(transitions[i].source, ctx->config)) {
    if (!BIT_HAS(i, conflicts)) {
        if (ctx->is_enabled(ctx, &transitions[i], ctx->event) > 0) {
            bit_or(conflicts, transitions[i].conflicts);
            bit_or(target_set, transitions[i].target);
            bit_or(exit_set, transitions[i].exit_set);
            BIT_SET_AT(i, trans_set);
            ctx->flags |= CTX_TRANSITION_FOUND;
            }
15      }
    }
}
```

To understand this piece of the algorithm, it is important to realize that the transitions in `transitions` are sorted from a post-order traversal of all `<transition>` elements in the original SCXML document. This corresponds to the *priority* of a transition from the definition for the optimal transition set in Sect. 10.2: Transitions with the same source state are given in document order and transitions within descendant source states precede those in ancestor source states. This means that the first enabled transitions (t_1) is necessarily in the optimal transition set and its *conflicts* will exclude all other transitions that cannot form an optimal transition set with t_1 included. By iterating all other transitions, we can successively establish the optimal transition set by adding enabled transitions that are not conflicting and skipping those that are.

We start the iteration (line 2) and skip any `<transition>` elements that originate in an `<initial>` or `<history>` element (line 3–4) as we will handle them differently when completing the target-set as the entry-set below. For the remaining transitions, we check whether they are *active*, *non-conflicting*, and *enabled* (line 6–8). The order of these conditions is arbitrary, though, the check for the enabled status of a transition is potentially expensive and thus the last condition.

If all these conditions hold for the current transition, we add its conflicts to the set of conflicting transitions, its targets to the intermediate target-set and its exit-set to the complete exit-set (line 9–11). Finally we remember the transition as being part of the optimal transition set in `trans_set` (line 12) and the fact that we found a transition at all as flag in the context (line 13).

Finally, we need to intersect the optimal transition set's exit set with the active configuration to arrive at the set of states actually exited (Fig. 10.1 **(3a)**) and determine whether we need to perform another round of spontaneous transitions or dequeue an event in the next round (Listing 10.7).

Listing 10.7 Establishing the actual exit-set and determining whether spontaneous transitions are exhausted

```
bit_and(exit_set, ctx->config);
if (ctx->flags & CTX_TRANSITION_FOUND) {
    ctx->flags |= CTX_SPONTANEOUS;
    ctx->flags &= ~CTX_TRANSITION_FOUND;
} else {
    ctx->flags &= ~CTX_SPONTANEOUS;
}
```

Now we have already established (1) the optimal transition set, (2) the exit set, and (3) an intermediate target set that we will have to complete below.

10.4.4 Remembering the History

Before we can establish the missing entry-set as the completion of the target-set, we will have to process any `<history>` elements (Fig. 10.1 **(3b)**) whose parent states are in the exit-set as they might eventually be entered within the same micro-step again (Listing 10.8).

Listing 10.8 Remembering the history

```
1   REMEMBER_HISTORY:
    for (i = 0; i < NUMBER_STATES; i++) {
      if (STATE_MASK(states[i].type) == STATE_HIST_SHALLOW ||
          STATE_MASK(states[i].type) == STATE_HIST_DEEP) {
5         if (BIT_HAS(states[i].parent, exit_set)) {
              bit_copy(tmp_states, states[i].completion);
              bit_and(tmp_states, ctx->config);
              bit_and_not(ctx->history, states[i].completion);
              bit_or(ctx->history, tmp_states);
10        }
      }
    }
```

If control flow reaches the inner-most block, *i* contains the index of a history state whose parent is about to be exited within the current micro-step and we have to remember its history. We defined a history state's completion as the set of states that are *covered* by the history and are not already covered by a nested history (note that a state can still be covered by more than one history elements with the same parent state). To remember a histories active states, we set all bits from the histories completion within the temporary state bit-array (line 6–7). Then, we intersect the states covered by the history with the active configuration and reset the context's history with the new history for the states covered (line 8–10).

Here, the fact that we excluded those states already covered by nested histories from the histories completion will ensure that no states covered by more deeply nested history elements are reset. If they are to be reset, we will pass the respective history element in a later step of the iteration.

10.4.5 Establishing the Entry Set

The next activity to perform is to complete the target-set as the actual entry-set (Fig. 10.1 **(3c)**). To this effect we, again, first define the complete entry set and later intersect it with the non-active states to arrive at the actual entry-set. The first thing to realize is that if a state (proper or otherwise) is in the target-set, all its ancestors will necessarily be active in the next configuration. As such, we can just add all ancestors of states in the target-set (Listing 10.9) and, subsequently complete them.

Listing 10.9 Extending the target set with all ancestors

```
ESTABLISH_ENTRY_SET:
bit_copy(entry_set, target_set);
for (i = 0; i < NUMBER_STATES; i++) {
    if (BIT_HAS(i, entry_set)) {
        bit_or(entry_set, states[i].ancestors);
    }
}
```

To complete the target-set and its ancestors more efficiently, we have to make sure that the states in the completion of a given state *s* always succeed *s* in document order. This seems obvious but is actually not necessarily the case if we targeted an `<initial>` or `<history>` pseudo-state as their completion might be siblings. As such, we have to postulate that for all children of a given parent, all `<initial>` elements precede `<history>` elements precede proper `<state>`s. This is merely a syntactic transformation of the SCXML document that we will have to perform prior to establishing the array with all state structures above. If this is given, we can iterate the set of all states in document order and dispatch on their type to add their completion to the entry set (Listing 10.10).

Listing 10.10 Adding the completion of all states into the entry set

```
for (i = 0; i < NUMBER_STATES; i++) {
    if (BIT_HAS(i, entry_set)) {
        // mask the MSB with the HAS_HIST flag
        switch (STATE_MASK(states[i].type)) {
            ...
        }
    }
}
```

The actual completion of a state from the preliminary uncompleted entry-set depends on its type as follows:

- `case STATE_PARALLEL:`

```
bit_or(entry_set, states[i].completion);
break;
```

  If a `<parallel>` element is in the entry set, all of its child states will have to be in the complete entry set.
- `STATE_INITIAL:`

```
for (j = 0; j < NUMBER_TRANSITIONS; j++) {
    if (transitions[j].source == i) {
        BIT_SET_AT(j, trans_set);
        CLEARBIT(i, entry_set);
        bit_or(entry_set, transitions[j].target);
        for (k = i + 1; k < NUMBER_STATES; k++) {
            if (BIT_HAS(k, transitions[j].target)) {
               bit_or(entry_set, states[k].ancestors);
            }
10       }
      }
  }
break;
```

If a transition or the completion of another state targeted an <`initial`> state, we search for its default <`transition`> and add the transition's target state and its ancestors to the complete entry-set. We do know that the initial transition's target state succeeds the initial state in document order (as we sorted the array of states accordingly), and can start the search for the target at the state succeeding the initial state (line 6).

- `STATE_COMPOUND:`

```
1 if (!bit_has_and(entry_set, states[i].children) &&
      (!bit_has_and(ctx->config, states[i].children) ||
       bit_has_and(exit_set, states[i].children)))
  {
5    bit_or(entry_set, states[i].completion);
     if (!bit_has_and(states[i].completion, states[i].children)) {
        for (j = i + 1; j < NUMBER_STATES; j++) {
          if (BIT_HAS(j, states[i].completion)) {
             bit_or(entry_set, states[j].ancestors);
10           break;
          }
        }
     }
  }
15 break;
```

When we encounter a compound state while completing the target set and its ancestors, we first have to check whether it is already complete (line 1–3) in which case we do not do anything. Otherwise, we add its completion and check (line 6) whether its completion is referencing a state more deeply nested (e.g., via an initial attribute into a non-child descendant), in which case we have to add the completion ancestors as well (line 8–11).

- `case STATE_HIST_SHALLOW:`
  `case STATE_HIST_DEEP:`

  Completing history states is the most complicated case as we have to account for various situations and take deep nested histories into account. We can differentiate two general cases first:

  – The history is empty:

```
1 for (j = 0; j < NUMBER_TRANSITIONS; j++) {
    if (transitions[j].source == i) {
      bit_or(entry_set, transitions[j].target);
```

```
     if(STATE_MASK(states[i].type) == STATE_HIST_DEEP &&
5        !bit_has_and(transitions[j].target, states[i].children))
     {
       for (k = i + 1; k < NUMBER_STATES; k++) {
         if (BIT_HAS(k, transitions[j].target)) {
           bit_or(entry_set, states[k].ancestors);
10         break;
         }
       }
     }
     BIT_SET_AT(j, trans_set);
15   break;
  }
}
```

If we never before exited the history's parent state, we merely need to add its default transition to the transition-set (to process its eventual executable content later) and the default transition's target to the entry-set. If the history is `deep`, its default *default history configuration* may be a descendant of a sibling, in which case we have to add its ancestors as well (line 4–13). For `shallow` histories, the standard mandates that the target is a sibling of the history.

– We already remembered states for the history:

```
1 bit_copy(tmp_states, states[i].completion);
  bit_and(tmp_states, ctx->history);
  bit_or(entry_set, tmp_states);
  if (states[i].type == (STATE_HAS_HIST | STATE_HIST_DEEP)) {
5   for (j = i + 1; j < NUMBER_STATES; j++) {
      if (BIT_HAS(j, states[i].completion) &&
          BIT_HAS(j, entry_set) &&
          (states[j].type & STATE_HAS_HIST)) {
        for (k = j + 1; k < NUMBER_STATES; k++) {
10        if (BIT_HAS(k, states[j].children) &&
              (STATE_MASK(states[k].type) == STATE_HIST_DEEP ||
               STATE_MASK(states[k].type) == STATE_HIST_SHALLOW)) {
            BIT_SET_AT(k, entry_set);
          }
15      }
      }
    }
  }
```

In this case, we need to add the states we remembered earlier which are covered by the history to the entry-set (line 1–4). If the current history element has nested history elements (line 5) and their parents were added to the entry-set via our coverage (line 7–9), we need to add them as well, to be processed likewise in a later iteration (line 11–15). Here, we can again exploit the fact that they will necessarily succeed the current history element in document order and start iteration at the state succeeding the current history pseudo state.

Now we have all the sets in place to perform the actual transitions and call executable content in the following sections.

10.4.6 Exiting States

Listing 10.11 Exiting states in reverse document order

```
1  size_t i = NUMBER_STATES;
   while(i-- > 0) {
     if (BIT_HAS(i, exit_set) && BIT_HAS(i, ctx->config)) {
       if (states[i].on_exit != NULL) {
5        err = states[i].on_exit(ctx, &states[i], ctx->event);
         if (err != ERR_OK)
           return err;
       }
       CLEARBIT(i, ctx->config);
10   }
   }
```

To exit states during a microstep (Fig. 10.1 **(4a)**) is merely to iterate all states from the complete exit-set (line 1–2) that are active (line 3) in reverse document order, invoke their `on_exit` handlers (line 4–8), and remove them from the active configuration (line 9).

10.4.7 Taking Transitions

Listing 10.12 Taking transitions in document order

```
1  for (i = 0; i < NUMBER_TRANSITIONS; i++) {
     if (BIT_HAS(i, trans_set) &&
         (transitions[i].type & (TRANS_HIST | TRANS_INITIAL)) == 0) {
       if (transitions[i].on_transition != NULL) {
5        err = transitions[i].on_transition(
                 ctx,
                 &states[transitions[i].source],
                 ctx->event);
         if (err != ERR_OK)
10         return err;
       }
     }
   }
```

After we exited all states from the intersection of the complete exit-set with the currently active configuration, we need to perform any eventual executable content associated with transitions in the optimal transition set in document order

(Fig. 10.1 **(4b)**). We do iterate the array with the transition structures in a post-order sequence, though, the optimal transition subset of all transitions is implicitly ordered in document-order. This becomes clear if we consider that for a transition t_1 in the optimal transition set, no other transition t_2 in the optimal transition set can precede t_1 in post-order and succeed t_2 in document-order as it would never be optimally enabled with its source state being ancestrally related to the source of t_1.

In this step, we will not yet perform executable content associated with transitions whose parent is an `<initial>` or `<history>` element (line 3) as these are to be processed after the `<onentry>` elements of their parent states.

10.4.8 *Entering States*

As the last activity within a micro-step, we need to enter all states from the intersection of the complete entry-set with the negated active configuration (Fig. 10.1 **(4c)**). There are, however, quite some additional activities associated with the entering of states that are outlined in Listing 10.13 and detailed below.

Listing 10.13 Entering states in document order.

```
for (i = 0; i < NUMBER_STATES; i++) {
  if (BIT_HAS(i, entry_set) && !BIT_HAS(i, ctx->config)) {
    if (STATE_MASK(states[i].type) == STATE_HIST_DEEP ||
      STATE_MASK(states[i].type) == STATE_HIST_SHALLOW ||
        STATE_MASK(states[i].type) == STATE_INITIAL)
      continue;
    BIT_SET_AT(i, ctx->config);
    // 1. Initialize data
    // 2. Perform executable content for on_entry
    // 3. Process history and initial transitions
    // 4. Raise done events
  }
}
```

1. **Initialize Data**
   After we added the new state to the active configuration, we need to initialize its associated `<data>` elements if the document has a `late` data binding. We do keep a bit-array of states that were already initialized in the interpreter's context and did transform all `<data>` elements accordingly.

Listing 10.14 Initializing nested data elements for late data binding

```
if (!BIT_HAS(i, ctx->initialized_data)) {
    if (states[i].data != NULL && ctx->exec_content_init != NULL) {
        ctx->exec_content_init(ctx, states[i].data);
    }
    BIT_SET_AT(i, ctx->initialized_data);
}
```

(2) **Perform Executable Content**
To perform the executable content is merely to invoke the states `on_entry` callback function for the generated code as introduced in Sect. 10.3.5.

Listing 10.15 Calling executable content for the entry of states

```
if (states[i].on_entry != NULL) {
  err = states[i].on_entry(ctx, &states[i], ctx->event);
  if (err != ERR_OK)
    return err;
}
```

(3) **Process History and Initial Transitions**
When we completed the target-set as the entry-set above, we did remember all initial and history transitions that would have to be performed, but ignored them when we performed the transitions' executable content after exiting the states from the exit-set above. Their respective bits are still set in the transition-set and the standard mandates to invoke their executable content after the parent states on-entry handlers.

Listing 10.16 Calling executable content for history and initial transitions

```
for (j = 0; j < NUMBER_TRANSITIONS; j++) {
  if (BIT_HAS(j, trans_set) &&
      (transitions[j].type & (TRANS_HIST | TRANS_INITIAL)) &&
      states[transitions[j].source].parent == i) {
    if (transitions[j].on_transition != NULL) {
      err = transitions[j].on_transition(ctx,
                                        &states[i],
                                        ctx->event));
      if (err != ERR_OK)
        return err;
      }
    }
  }
```

(4) **Raise Done Events**
Special considerations have to be given when entering `<final>` states as part of a microstep.

```
if (STATE_MASK(states[i].type) == STATE_FINAL) {
  ...
}
```

If the parent of the final state is the `<scxml>` element itself, the interpreter is done and we set the `CTX_TOP_LEVEL_FINAL` flag in the interpreter's context (Listing 10.17).

Listing 10.17 Top-level final state reached

```
if (states[i].ancestors[0] == 0x01) {
    ctx->flags |= CTX_TOP_LEVEL_FINAL;
  }
```

Otherwise, if the final state is the child of a compound state, we need to raise a `done.state.[ID]` event on the interpreter's internal queue and attach any eventual `<donedata>` with the event (Listing 10.18).

Listing 10.18 Final state of a compound state entered

```
else {
     const elem_donedata* donedata = &elem_donedatas[0];
     while(ELEM_DONEDATA_IS_SET(donedata)) {
       if unlikely(donedata->source == i)
         break;
       donedata++;
     }
    ctx->raise_done_event(ctx,
                              &states[states[i].parent],
                 (ELEM_DONEDATA_IS_SET(donedata) ? donedata : NULL));
  }
```

In this last case, we also need to check whether the current final state is the last one to finalize all children of a parallel ancestor, in which case we need to raise a `done.state.[PARALLEL_ID]` event in addition (Listing 10.19).

Listing 10.19 Raising done events for finalized parallel states

```
1  for (j = 0; j < NUMBER_STATES; j++) {
     if (STATE_MASK(states[j].type) == STATE_PARALLEL &&
         BIT_HAS(j, states[i].ancestors)) {
       bit_and_not(tmp_states, tmp_states);
5      for (k = 0; k < NUMBER_STATES; k++) {
         if (BIT_HAS(j, states[k].ancestors) &&
             BIT_HAS(k, ctx->config)) {
           if (STATE_MASK(states[k].type) == STATE_FINAL) {
             bit_and_not(tmp_states, states[k].ancestors);
10         } else {
             BIT_SET_AT(k, tmp_states);
           }
         }
       }
15     if (!bit_any_set(tmp_states)) {
         ctx->raise_done_event(ctx, &states[j], NULL);
       }
     }
   }
```

We start by iterating all states and search for parallel states which are ancestrally related to the current final state (line 1–3). If we found such a state, we clear out the temporary bit array of states to remember any active descendant of the parallel (line

5–7). If we found an active state in the descendants of the parallel state under consideration and it is a final state itself, we clear all its ancestors in the temporary bit-array, if it is anything else, we set its ancestor's bits. After we processed all active descendant states of the parallel in this manner and the temporary bit-array is empty (line 15–17), all of the parallel's child states have also entered a final state and we need to raise the `done.state.[PARALLEL_ID]` event for the parallel state.

This concludes the description of the `microstep(T)` algorithm in ANSI C and we will evaluate its performance and memory consumption in the following sections.

10.5 Evaluating the ANSI C Implementation

In this section, we will evaluate the ANSI C algorithm presented above with regard to its runtime, binary size, and memory consumption. As a baseline, we took the `microstep(T)` implementation from our `uSCXML` implementation, which is relevant as it, rather literally, employs the pseudo-code from Appendix D of the SCXML recommendation. Though, even with this baseline implementation, we already employ some caching, e.g. for state look-ups by identifier, the exit- and target-set of transitions and proper ancestors of two states. As such, it establishes a lower bound for any implementation that approaches the `microstep(T)` algorithm as specified in the recommendation. We are aware that it was never the intention of said pseudo-code to be performant or small, but many SCXML interpreters do, indeed, implement the `microstep(T)` algorithm very similarly.

10.5.1 Methodology

For all our measurements, we transformed all SCXML IRP tests for the ECMAScript datamodel and generated the compound data structures as introduced above. We wrote the callback functions as required for the SCXML context connecting to the respective functionality in the `uSCXML`[2] interpreter and explicitly excluded:

- 37 tests due to missing support for the `<invoke>` element.
- 17 tests due to missing support for anything but the SCXML I/O processor.
- 4 tests that attempt to retrieve data from a URL.
- 1 test with an XML node in a variable.
- Some more manual tests.

[2] https://github.com/tklab-tud/uscxml (accessed January 26th, 2015).

This set of tests forms the basis for all subsequent measurements below. All measurements are taken on a MacBook 13" (early 2015) with Intel Core i7 CPU @ 3.1 GHz. While this is not exactly a *resource constrained device*, the actual values measured give every reason to assume that the implementation is perfectly suited for resource constrained devices.

10.5.2 Compliance

The set of SCXML IRP tests that is passed by our implementation for the ECMAScript data-model is given in Table 10.1. Do note that we pass all tests for core constructs but the one for invocation order (`test422`) as we did not implement `<invoke>` yet. Even though the tests are merely an enumeration of correct behavior for a compliant interpreter and no proof of compliance, it is a good indicator of a *largely correct* implementation of `microstep(T)`.

We also wrote initially failed and ultimately passed three additional tests for deep completions via the initial attribute and nested history pseudo-states (deep and shallow) to account for some border cases we realized when designing the algorithm above.

Table 10.1 Number of tests in the SCXML Implementation and Report Plan with corresponding section from specification

Class	#Pass	#Total	Class	#Pass	#Total
Core constructs		*40 (1)*	*Data model and manipulation*		*50 (4)*
General	2	2	Data	5	7
State	1	1	Assign	4	4
Final	2	2	Donedata	1	1
OnEntry	2	2	Content	3	3
OnExit	2	2	Param	3	3
History	4	4	Script	3	4 (1)
Events	4	4	Expressions	7	8 (3)
Transition selection	22	23 (1)	System variables	19	20
Executable content		*13*	*External communications*		*51 (3)*
Raise	1	1	Send	16	19 (1)
If	3	3	Cancel	2	3
Foreach	7	7	Invoke	0	29 (2)
Evaluation	2	2	*Data models*		*51*
Event I/O processors		*28 (1)*	NULL	1	1
SCXML	10	16	ECMAScript	15	20
Basic HTTP	0	12 (1)	XPath	0	30
			Total	140	*233 (9)*

Brackets indicate manual tests

10.5.3 Performance

For the performance measurements, we instrumented the code-base with timers using `mach_absolute_time` as the highest precision monotonic clock available. It is difficult to get any reliable information about its precision, accuracy or resolution. However, an example in the official technical QA1398[3] from Apple does convert its return value into nanoseconds, suggesting a sufficient granularity for the measurements. Furthermore, all measurements were averaged over 1.000 iterations and the methodology was the same for the baseline. Still, the approach of measuring the performance of a given piece of code by averaging its runtime is far from objective as seemingly unrelated changes in the runtime can have a considerable effect on the measurements [7]. As such, the numbers below are to be interpreted with some reservations.

Using this approach, we were able to measure the performance of 132 individual tests, with the remaining 8 tests relying on the timeout of an event, which prevented us from measuring. We did measure the time to completion for a single interpretation per test excluding and subtracted the time spent in the data-model's functions. The difference was divided by the number of iterations for the `microstep(T)` algorithm described above.

Figure 10.7 depicts a distribution for the average duration of such an iteration per SCXML IRP test with 5 us bins. We can see that for the majority of tests, their microsteps averaged to about 5–15 us, which translates to `650.000 - 2.000.000`

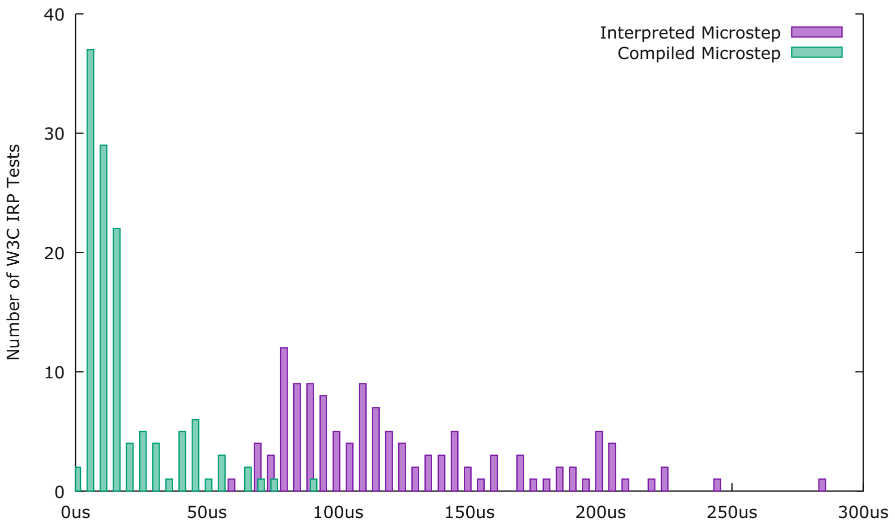

Fig. 10.7 Distribution of the execution speed of a single microstep for the interpreted and compiled case (averaged per SCXML IRP test)

[3] https://developer.apple.com/library/mac/qa/qa1398/_index.html (accessed January 26th, 2016).

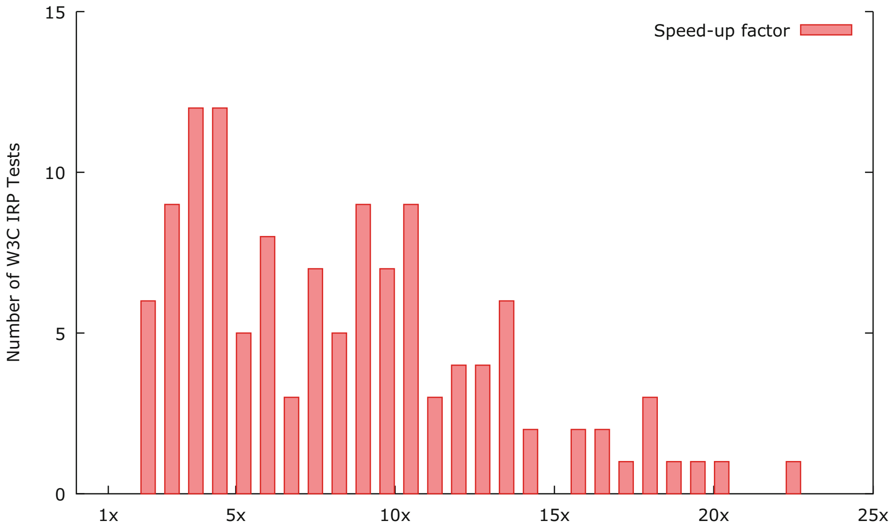

Fig. 10.8 Distribution of the speed-up for a single microstep (averaged per SCXML IRP test)

iterations per second or, at most, `300.000-1.000.000` events per second (when assuming no spontaneous transitions).

We also did a direct comparison for the average duration of a microstep iteration per test and Fig. 10.8 depicts a distribution of the speed-up factor when using the algorithm described above compared to the baseline. We can see that the proposed algorithm always outperforms a more literal implementation of the pseudo-code from Appendix D in the SCXML recommendation and, on occasion, is more than 20 times as fast.

10.5.4 Binary Size and Memory

An important consideration when targeting a resource constrained platform is the memory available. For example, the ATmega8 from Atmel only features 8 KB of flash memory with 512 Byte SRAM for dynamic data, its more powerful counterparts up to 256 KB flash memory with 8 KB SRAM. As such, a compact representation for the logic representing the control flow from the SCXML document has a direct consequence for its applicability in this domain. A major problem in this regard is the employed data-model: A single instance of the JavaScriptCore ECMAScript implementation will, regardless of actual usage, allocate 8 MB of memory upon instantiation on top of its already considerable binary size; orders of magnitude more than what would be available on an ATmega8. One scripting language explicitly touted for scripting on resource constrained devices is Lua with a binary size of round about 80 KB and very conservative memory usage and, indeed, the `uSCXML` platform does support a Lua datamodel.

While the SCXML recommendation does provide a normative specification for an ECMAScript data model and a supplementary W3C note for an XPath data model, there is no mandatory requirement for a compliant interpreter to implement either. This offers considerable flexibility, but comes with the cost of reduced interoperability.

For our measurements, we explicitly excluded the size of the data-model. If one were to seriously target a resource constrained platform, ultimately, a data-model that can syntactically transformed onto ANSI C seems most suited as it can directly be subjected to the compiler for the respective platform without any requirement for runtime interpretation. As such, we only measured the size of the compiled control flow logic with all required static data and executable content functions introduced above and excluded anything linked from the user-supplied callback functions. The distribution of binary sizes for the 140 IRP tests from Table 10.1 is given in Fig. 10.9.

Two distributions for different compiler switches are displayed. When optimizing for speed (`-Ofast`), the resulting binaries will be anywhere from 3 to 6 KB. When optimizing for size, the resulting binaries are round about the same size, at times even somewhat larger. With the possible options for reducing the required memory by dropping some of the bit-arrays in the static data structures introduced in Sects. 10.3.2 and 10.3.1, this size is perfectly suited to run on a device with 8–16 KB of memory. However, as it is, the size for compiled binaries grows quadratic with the size of the input SCXML document as each additional state or transition will increase each relation modeled in the bit-arrays (cf. Figs. 10.3 and 10.5).

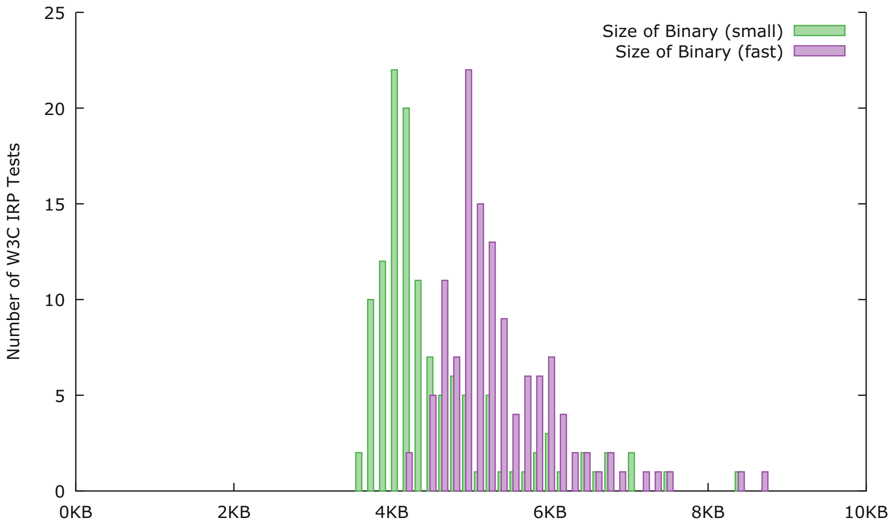

Fig. 10.9 Distribution of the 16-bit binary size for the compiled SCXML IRP tests (state-transitioning and executable content calls only, horizontal lines denote base size for *empty* statechart)

With regard to the dynamic memory requirements at runtime, it is noteworthy that the algorithm above does, at no point, allocate memory on a heap structure (`malloc`) and all allocations are performed on the stack. The sum of memory required per iteration depends on the number of states and transitions in the original SCXML document. If we assume an original SCXML document with 50 states and transitions each, we can calculate its dynamic memory requirements as follows: Each instance of an interpretation will require round about 90 bytes for its SCXML context structure (Fig. 10.6) and every iteration of a microstep will instantiate

- **3 unsigned integer variables** as indices during iteration for a total of 6 bytes on a 16-bit architecture.
- **2 bit-arrays for transitions set**, namely for the optimal transition set in `trans_set` and for conflicting transitions in `conflicts`, amounting to 2 ×`ceil`(NUMBER_TRANS / 8) bytes.
- **4 bit-arrays for transitions set**, namely for the set of states targeted by transitions in the optimal transition set as `target_set`, the entry-set in `entry_set` and the exit-set in `exit_set`. One more bit array is allocated as a general, temporary bit-array `tmp_states` and used to remember and reenter history states and when raising the `done.state.[PID]` event for parallel states. This amounts to a total of 4 ×`ceil`(NUMBER_STATES / 8) bytes.
- **A single additional byte** for the return value in `err`.

If we, again, assume a state-chart with 50 transitions and states, any bit-array will consist of 7 bytes for a total of 49 bytes allocated on the stack per invocation (not accounting for alignment padding). The development for the memory requirements of static and dynamic memory is depicted in Fig. 10.10, excluding memory for code and additional elements other than transitions and states.

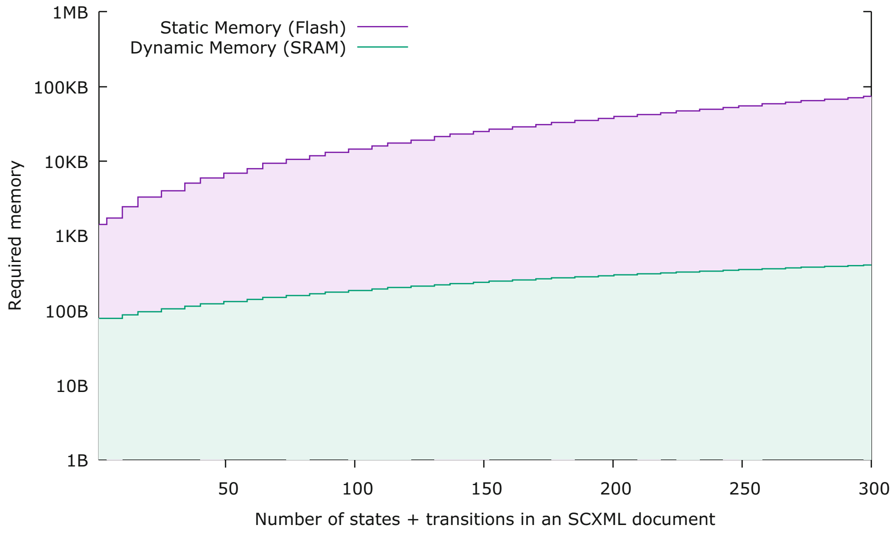

Fig. 10.10 Static memory required for the data structures (transitions and states) and dynamic memory (context and microstep stack)

10.6 Transformation for VHDL

While the C implementation of `microstep(T)` described above will already allow to address a wide range of off-the-shelf micro-controllers, a single iteration will still require tens of thousands of cycles. In this section we present a hardware realization for a subset of SCXML state-charts and discuss its possible performance. To this effect, we will not generate C code, but descriptions for hardware building blocks, expressed in the widely used hardware description language VHDL. With VHDL, it is possible to program FPGA logic blocks for dynamic hardware state machines and even to design custom ASICs.

The general description is, again, aligned with the set of steps depicted in Fig. 10.1 and based on the pre-calculated sets and relations already introduced as part of the C implementation above. While we already excluded the `<invoke>` element, custom I/O processors and some other features for the description of the C algorithm, the domain of the VHDL transformation will be even more restricted:

- We will not concern ourselves with any data-model, but only describe the transitioning of active configurations and the corresponding entry-, exit-, and transition-sets.
- We do not yet address the semantic of the `<history>` element nor are `<initial>` elements supported. The `initial` attribute is supported though.
- Events are enumerated and expressed as individual lines. Any data attached to an event would be inaccessible anyway as we do not provide a data-model.
- No executable content other than `<raise>` and `<send>` are supported and these can only address simple events to either the internal or external event queue.

The general architecture of the hardware realization is depicted in Fig. 10.11 and consists of a *microstepper* with an attached *event controller*. Each microstep is performed in a single cycle and several outputs are available

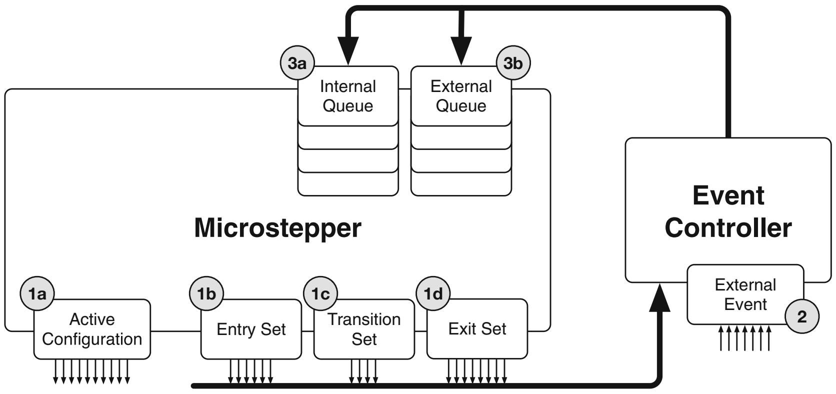

Fig. 10.11 A high level overview of the generated hardware architecture

- The **Active Configuration (1a)** provides the valuation of active states in the current configuration. It's realized as a parallel bus, where every line signifies the activation status of one state, indicating an active state with `HIGH` and an inactive state with `LOW`. The bus width equates to the number of proper states in the SCXML document.
- The **Entry Set (1b)** pins provide the information, which states were in the entry-set when the microstepper transitioned to the active configuration by setting the corresponding pin to signal `HIGH`. To save some of the rare I/O pins, we just generate pins for states, that have defined an `<onentry>` child element.
- Analogously, the **Exit States (1d)** pins provide the information, which states were in the micro-step's exit-set by setting the corresponding pin to signal `HIGH`. Again, we just generate pins for states, that have an `<onexit>` child element.
- The **Transition Set (1c)** pins provide the information, which `<transition>` elements with executable content were in the optimal transition set.
- For the **Internal Queue (3a)** and the **External Queue (3b)**, the microstepper offers a writing interface, that provides enough pins to differentiate the individual events specified in the SCXML document.

  Since we cannot, in the general case, give an upper-bound for the maximum length of either event queue at transformation time [8], it is important to take carenot to overflow them. If an event is about to be enqueued on a full queue, the microstepper will, for now, just assume an error state readable through an interface pin.

The event-controller will, depending on the occurrence of `<raise>` and `<send>` elements in executable content, deliver these events in accordance with the valuation of the entry-, exit-, and transition-set bus. It is also available to, asynchronously, deliver additional external events not enqueued by the state-chart itself **(2)**.

Figure 10.12 illustrates the inside architecture of the microstepper component. It mainly consists of the event queues **(3a–b)** and an elaborate Moore state machine **(4a–c)**, to perform the actual micro-steps. The most relevant parts of the state machine are the transition logic **(4a)** and the state memory **(4b)**. These are described more detailed in the following chapter. As the current state configuration and relevant sets are available as interface busses, the output logic **(4c)** is just responsible for setting the `completed` signal, which indicates that the state-chart is in a top-level final state.

With the general architecture of the hardware established, we can now describe its actual implementation to realize the steps from Fig. 10.1 in more detail.

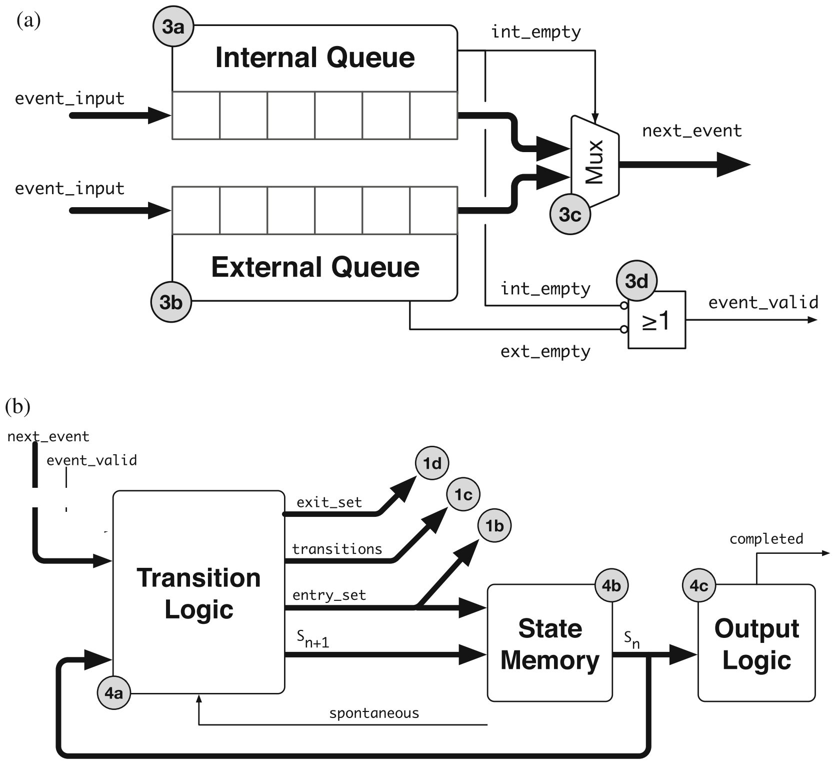

Fig. 10.12 Architecture of the microstepper. (**a**) Internal and external event queue with bus selection. (**b**) Finite-State-Machine Implementation

10.6.1 Dequeuing Event

The logic depicted in Fig. 10.12 **(3a–d)** shows how non-null event dequeuing is implemented: If there is an event enqueued at the internal queue, the `int_empty` signal is `LOW` and the multiplexer **(3c)** connects the `next_event` bus to the internal queue or external queue otherwise. If both the `int_empty` and `ext_empty` signals are `HIGH`, the `event_valid` signal goes to `LOW` to indicate that no events are available. Both signals along with the `spontaneous` signal from the state memory can, subsequently, be used to perform transition selection.

10.6.2 Selecting Transitions

Just as with the C implementation, the next step is to establish the optimal transition set for the current event (Fig. 10.13). We have already described, in the scope of the

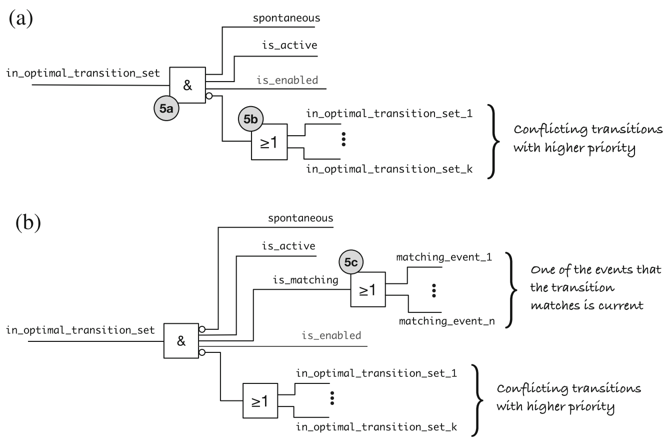
(a)
spontaneous
is_active
in_optimal_transition_set
&
is_enabled
5a
5b
≥1
in_optimal_transition_set_1
in_optimal_transition_set_k
Conflicting transitions with higher priority
(b)
spontaneous
is_active
5c
matching_event_1
is_matching
≥1
One of the events that the transition matches is current
matching_event_n
in_optimal_transition_set
&
is_enabled
in_optimal_transition_set_1
≥1
in_optimal_transition_set_k
Conflicting transitions with higher priority

Fig. 10.13 Establishing the optimal transition set. (**a**) Transition selection for spontaneous transitions. (**b**) Transition selection for non-spontaneous transitions

C implementation above, how we can employ a post-order traversal of all transitions to have higher priority transitions precede those with a lower priority. Furthermore, we did introduce a *conflicts* relation of transitions to prevent the selection of invalid transition sets. Both are also relevant to select the optimal transition sets with dedicated hardware.

If the last micro-step did not exhaust spontaneous transitions, the `spontaneous` is still set to `HIGH` in the state memory and the logic in Fig. 10.13a is applied. For any given spontaneous transition, we will set its `in_optimal_transition_set` line to `HIGH` **(5a)** if its parent state is active as per configuration in the state memory and no other spontaneous transition with a higher priority conflicts **(5b)**. Here, we can just connect all `in_optimal_transition_set` lines for conflicting transitions with a higher priority as they are known at transformation time. We also included an eventual `is_enabled` signal, which would need to be set by some external component that would realize the data-model.

The case for non-spontaneous transitions, selected for a non-null event, is very similar, but an enumeration of matching events would need to set the `is_matching` signal **(5c)** to `HIGH` as well. This will give us the valuation of signals for the external interface bus at **(1c)** above.

10.6.3 Establishing the Exit-Set

When we identified the set of `in_optimal_transition_set` signals that are set to `HIGH` for transitions in the optimal transition set, we can instantaneously establish the exit-set of the current micro-step. In the scope of the C implementation, we argued that we can identify a transition's complete exit-set as the exit-set

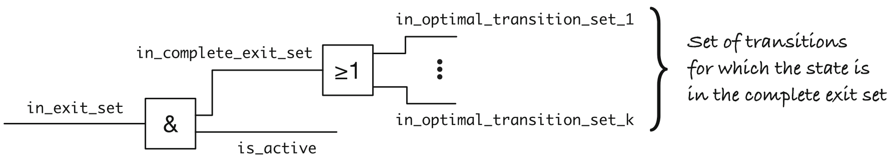
in_optimal_transition_set_1
in_complete_exit_set
≥1
in_exit_set
&
in_optimal_transition_set_k
is_active
Set of transitions for which the state is in the complete exit set

Fig. 10.14 Establishing the exit-set by intersecting the complete exit-set with the active configuration

when assuming the complete configuration. Now, if we intersect the complete exit-set with the active configuration, we arrive at the micro-step's actual exit-set (Fig. 10.14) and can set the respective signals to high on the external interface bus **(1d)**.

10.6.4 Establishing the Entry-Set

As we do not support `<history>` elements yet, the next step is to establish the entry-set of the current micro-step. This is by far the most complicated step, but by regarding the implementation in C, we can gain some insights that help us to understand the respective logic.

In the C implementation, there were three general situations for any given proper state to become part of the complete entry set:

1. The state is **targeted directly** by a transition in the optimal transition set.
2. The state is added as an ancestor of a targeted state (**ancestor completion**).
3. The state is added as the completion of a parent state (**descendant completion**).

If any state is targeted directly, it will set its `in_complete_entry_set_up` signal to `HIGH`, which causes ancestor completion for all its ancestor states (Fig. 10.15a, b). This signal is received by composite parent states and recursively passed to their parents causing all targeted states and their ancestors to have the respective signal set to `HIGH`. In order to arrive at a valid completion, any composite states added via ancestor completion will have to be completed as well (descendant completion). Composite parents of type `PARALLEL` will, unconditionally, add all their child states to the complete entry set (Fig. 10.15c). Composite states of type `COMPOUND` are more complicated: They will only need to be completed if they are not already complete, that is, if none of their children are already active and not exited during the current micro-step (Fig. 10.15d) and the given child state is the default completion per document order or `initial` attribute.

This will, recursively, establish the complete entry set which has to be intersected with the set of states that are active and not in the exit-set (Fig. 10.16) to arrive at the actual entry set for the external interface bus at **(1b)**.

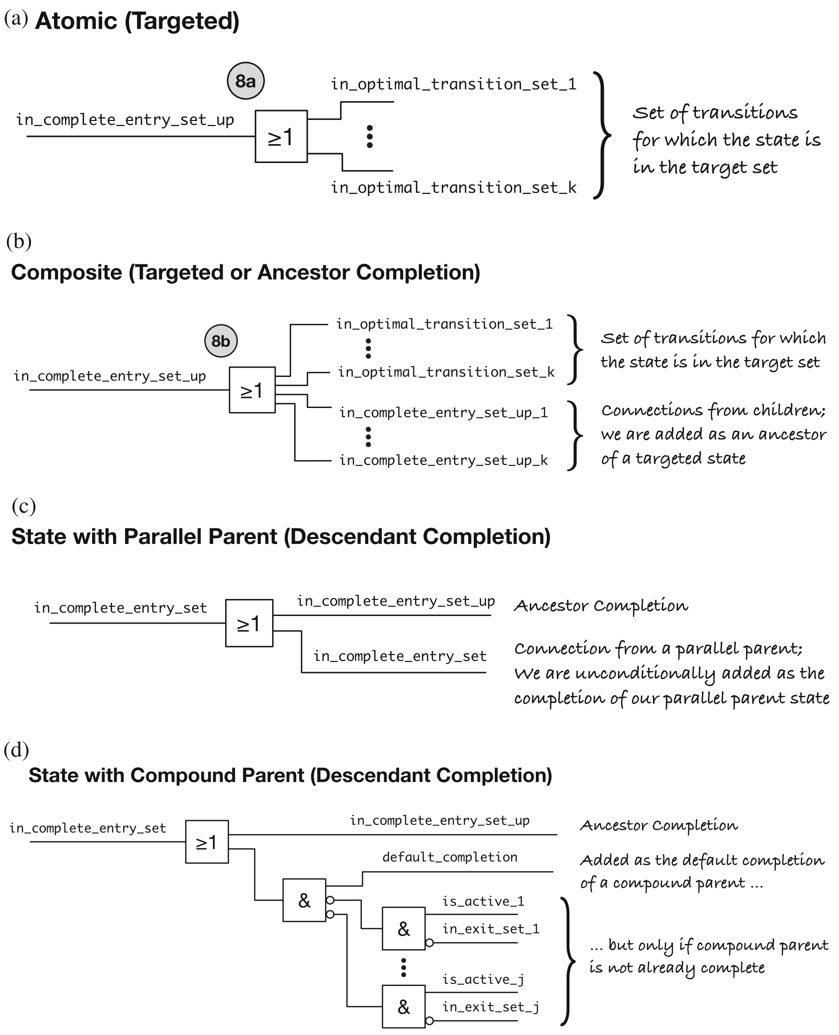
(a) Atomic (Targeted)
8a
in_complete_entry_set_up
≥1
in_optimal_transition_set_1
in_optimal_transition_set_k
Set of transitions for which the state is in the target set
(b) Composite (Targeted or Ancestor Completion)
8b
in_complete_entry_set_up
≥1
in_optimal_transition_set_1
in_optimal_transition_set_k
Set of transitions for which the state is in the target set
in_complete_entry_set_up_1
in_complete_entry_set_up_k
Connections from children; we are added as an ancestor of a targeted state
(c) State with Parallel Parent (Descendant Completion)
in_complete_entry_set
≥1
in_complete_entry_set_up
Ancestor Completion
in_complete_entry_set
Connection from a parallel parent; We are unconditionally added as the completion of our parallel parent state
(d) State with Compound Parent (Descendant Completion)
in_complete_entry_set
≥1
in_complete_entry_set_up
Ancestor Completion
default_completion
Added as the default completion of a compound parent ...
&
is_active_1
in_exit_set_1
is_active_j
in_exit_set_j
... but only if compound parent is not already complete

Fig. 10.15 Completing the target set as the complete entry set. (**a**) An atomic state added by being targeted directly. (**b**) A composite state added by being targeted directly or via ancestor completion. (**c**) A state added by its parallel parent state. (**d**) A state added as the default completion of a compound parent state

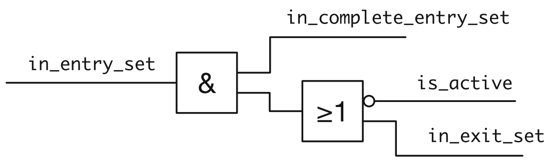

Fig. 10.16 Establishing the entry-set by intersecting the complete entry-set with the subset of the active configuration that is not exited

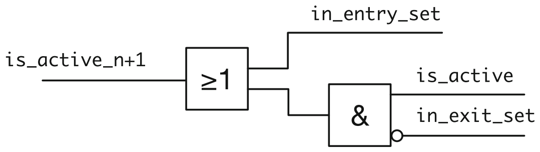

Fig. 10.17 The new active configuration is the set of states already active and not exited together with the set of states entered

10.6.5 Observable Performance

In the C implementation above, the preparations above triggered the actual exiting, transitioning, and entering of states. For the hardware implementation, these activities are to be performed by components connected to the external interface bus and dispatching over the various sets. As such, we just need to set the follow-up configuration (Fig. 10.17) as the set of states already active and not exited together with the set of states entered in the state-memory and process the next microstep.

10.6.6 Evaluation

The transformation from SCXML to VHDL is still very raw with only a select few language features implemented. As such, it is futile to evaluate its compliance with regard to the SCXML IRP tests. We did, however, write two simple SCXML statecharts that we successfully simulated to pass.

The first test in Listing 10.20 employs a parallel state with an atomic and a compound child state, with the compounds child, in turn, an atomic event raising an event upon entry on the internal queue that matches a transition in the other atomic state.

Listing 10.20 VHDL test for a parallel state with nested compound state

```
<parallel id="p1">
    <state id="p1.1">
       <transition event="foo" target="pass"/>
    </state>
   <state id="p1.2">
   <state id="p1.2.1">
  <onentry>
    <raise event="foo" />
    </onentry>
    </state>
  </state>
 <final id="pass" />
 <final id="fail" />
</parallel>
```

The second test in Listing 10.21 relies on transition preemption of the first transition by the more deeply nested second transition to pass.

Listing 10.21 VHDL test for transition preemption

```
<state id="s1">
  <transition event="foo" target="fail"/>
  <state id="s1.1">
    <onentry>
      <raise event="foo" />
    </onentry>
    <transition event="foo" target="pass"/>
  </state>
  <final id="pass" />
  <final id="fail" />
</state>
```

Ultimately, it is definitely desirable to get a more rigorous evaluation of the VHDL description's compliance and we are confident that, by aligning the VHDL description with the C implementation, we will be able to pass a similar subset.

10.6.6.1 Performance

Since all dynamic functions such as transition-, entry-, and exit-set generation or the calculation of the next configuration are implemented via combinatorial logic, the hardware performs one microstep per clock cycle. The maximum frequency for such a hardware component depends on several properties of the original SCXML state-chart and the hardware employed:

- **Critical Path:** The critical path is the longest combinatorial path in the design. A clock cycle has to be long enough, for a signal to pass through this path and stabilize. In our implementation it highly depends on the interleaving depth of the state machine and, as such, the complexity of the original SCXML document.
- **Hardware Specifics:** In particular the signal propagation time, which depends, among other things, on fabrication node and core voltage, is an important factor for the pass-through time of the longest path.

We expect, in any case, that the state controller's speed will not be the limiting factor for the overall system, external components like sensors and actuators are orders of magnitude slower and would stall the microstepper most of the time.

10.6.6.2 Hardware Costs

If we are to mold the VHDL description into an ASIC, we need to estimate the chip area required for the various transistors in our solution. Since this number depends on many factors we present a worst case estimation, wherein we treat every state in the SCXML document as an atomic state, which is the default completion of a `COMPOUND` state. We will further assume all of these states to have two transitions event driven transitions, and to be the target of two transitions each. This hypothetical set of states represents the worst "transistors per transition" relation.

From this assumptions we get

- `04` transistors for the buffer to save the state,
- `16` transistors for the atomic state function,
- `06` transistors for the activation set of the atomic state,
- `12` transistors for the exit set and the exclusive exit line,
- `06` transistors for the additional pins at the parent gates,
- `18` transistors for the transition function,
- `16` transistors for the interface buffers for the sets on the external interface bus,

for a total of 78 transistors per state. For a comparison, an Intel i7 Haswell-E has around 2.6 billion transistors.

If we are to implement the VHDL on an FPGA we need to estimate the required logic cells and flip-flop memory cells. Since an FPGA can build logic cells via its architecture, most vendors give "logic cell equivalent" numbers for their products. For our assumed worst case scenario above, we get

- `01` flip-flop to save the state,
- `03` gates for the atomic state function,
- `01` gates for the activation set of the atomic state,
- `02` gates for the exit set and the exclusive exit line,
- `03` gates for the additional pins at the parent gates,
- `05` gates for the transition function,
- `05` flip-flops for the interface buffers for the sets on the external interface bus,

for a total of 20 logic cells per state. As a consequence, a Xilinx Spartan 6 SLX9 would hold about 450 states. We choose this FPGA as comparison, because it is the smallest FPGA which could hold the AX8 as a VHDL description of the AVR architecture, which would be able to run the C implementation.

10.7 Conclusion

In this chapter we presented two implementation of the `microstep(T)` algorithm, central to every SCXML interpreter, one in ANSI C, another in VHDL. In the scope of the ANSI C implementation, we introduced several sets and relations that can be derived syntactically from a given SCXML document along with a few important observations:

- Most of the criteria for an optimal transition sets can be derived syntactically and encoded in a static $\texttt{conflicts}(t_1, t_2) \subseteq T \times T$ relation to identify pairs of transitions that can, for whatever reason, never be part of the same optimal transition set.

- The post-order traversal sorting for transitions is equivalent to the *priority* of a transition. Together with the `conflicts(t₁,t₂)` relation, this allows to identify the optimal transition set in a single iteration of transitions with most steps skipped very early.
- The complete exit set of a transition, as a superset of the actual exit set can be calculated at transformation time. The actual exit set is the intersection of this complete exit set with the active configuration. This notion extends to sets of transitions, i.e. the optimal transition set.
- The same is true for the complete entry set and the actual entry set of a transition set.
- Sorting the states such that the states in a given state's completion will always succeed the given state in document order allows to identify the entry set in a single iteration after we identified the target sets' ancestors.
- All of a state-chart's history can be encoded in a single bit per state.

By exploiting these techniques, we were able to improve the performance for a `microstep(T)` implementation considerably. Along with a transformation of an SCXML document onto a set of native data-structures, we managed to provide semantically equivalent object code with a size suitable to be deployed for even the smallest of micro-controllers.

The insights gained from the ANSI C implementation were subsequently applied for a transformation from SCXML onto VHDL to implement SCXML as dedicated hardware elements, be it by programming FPGAs or even to mold custom ASICs on a die. Even though the VHDL transformation was not evaluated with the same scientific rigor as the ANSI C implementation, we are confident that it provides an excellent starting point to support a larger set of SCXML language features.

Acknowledgements This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. B0101-15-1292, Development of Smart Space to promote the Immersive Screen Media Service) and by the German Research Foundation (DFG) as part of the projects B02 within the Collaborative Research Center (CRC) 1053 – MAKI.

References

1. Barnett, J., Bodell, M., Dahl, D., Kliche, I., Larson, J., Porter, B., et al. (2012). *Multimodal architecture and interfaces*. W3C recommendation, W3C. http://www.w3.org/TR/2012/REC-mmi-arch-20121025/.
2. Berjon, R., Faulkner, S., Leithead, T., Pfeiffer, S., O'Connor, E., & Navara, E. D. (2014). *HTML5*. Candidate recommendation, W3C. http://www.w3.org/TR/2014/CR-html5-20140731/.
3. Crane, M. L., & Dingel, J. (2005). *On the Semantics of UML State Machines: Categorization and Comparison*. Technical Report 2005-501, School of Computing, Queen's.
4. Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming, 8*(3), 231–274.

5. Harel, D., Pnueli, A., Schmidt, J. P., & Sherman, R. (1987). On the formal semantics of statecharts (extended abstract). In *Proceedings of the Symposium on Logic in Computer Science*, Ithaca, NY, USA (pp. 54–64).
6. Hosn, R., Carter, J., Burnett, D., Lager, T., Barnett, J., Raman, T., et al. (2015). *State chart XML (SCXML): State machine notation for control abstraction*. W3C recommendation, W3C. http://www.w3.org/TR/2015/REC-scxml-20150901/.
7. Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices, 44*(3), 265–276.
8. Radomski, S. (2015). *Formal Verification of Multimodal Dialogs in Pervasive Environments*. Ph.D. thesis, Technische Universität Darmstadt. http://tuprints.ulb.tu-darmstadt.de/5184/
9. von der Beeck, M. (1994). A comparison of statecharts variants. In H. Langmaack, W. P. de Roever, & J. Vytopil (Eds.), *Formal techniques in real-time and fault-tolerant systems. Lecture notes in computer science* (Vol. 863, pp. 128–148). Berlin, Heidelberg: Springer.