Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A systematic review of code generation proposals from state machine specifications

Eladio Domínguez [a], Beatriz Pérez [b,*], Ángel L. Rubio [b], María A. Zapata [a]

[a] Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, E-50009 Zaragoza, Spain
[b] Dpto. de Matemáticas y Computación, Universidad de La Rioja, E-26004 La Rioja, Spain

ARTICLE INFO

Article history:
Received 9 August 2011
Received in revised form 27 April 2012
Accepted 30 April 2012
Available online 23 May 2012

Keywords:
UML state machines
Finite state machines
Statecharts
Code generation
Systematic review

ABSTRACT

Context: Model Driven Development (MDD) encourages the use of models for developing complex software systems. Following a MDD approach, modelling languages are used to diagrammatically model the structure and behaviour of object-oriented software, among which state-based languages (including UML state machines, finite state machines and Harel statecharts) constitute the most widely used to specify the dynamic behaviour of a system. However, generating code from state machine models as part of the final system constitutes one of the most challenging tasks due to its dynamic nature and because many state machine concepts are not supported by the object-oriented programming languages. Therefore, it is not surprising that such code generation has received great attention over the years.
Objective: The overall objective of this paper is to plot the landscape of published proposals in the field of object oriented code generation from state machine specifications, restricting the search neither to a specific context nor to a particular programming language.
Method: We perform a systematic, accurate literature review of published studies focusing on the object oriented implementation of state machine specifications.
Results: The systematic review is based on a comprehensive set of 53 resources in all, which we have classified into two groups: pattern-based and not pattern-based. For each proposal, we have analysed both the state machine specification elements they support and the means the authors propose for their implementation. Additionally, the review investigates which proposals take into account desirable features to be considered in software development such as maintenance or reusability.
Conclusions: One of the conclusions drawn from the review is that most of the analysed works are based on a software design pattern. Another key finding is that many papers neither support several of the main components of the expressive richness of state machine specifications nor provide an implementation strategy that considers relevant qualitative aspects in software development.

Contents

* Corresponding author.
E-mail addresses: noesis@unizar.es (E. Domínguez), beatriz.perez@unirioja.es (B. Pérez), arubio@unirioja.es (Á.L. Rubio), mazapata@unizar.es (M.A. Zapata).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.04.008

1. Introduction

Model Driven Development (MDD) [1,2] encourages the use of models in the software development process with the aim of generating the application source code through automatic model transformations. Following this approach, modelling languages are used during the design and analysis processes for diagrammatically model the artifacts that specify the structure and behaviour of systems. In particular, whereas class diagrams are the mainstay of object-oriented analysis and design for representing the static structure of a system, state machine specifications (including UML state machines [3], finite state machines [4] and Harel statecharts [5]) are considered the most widely used method to specify the dynamic behaviour of reactive systems.

The models describing the different aspects of a system are taken as a starting point to finally develop the software system. At this point, it is well known that the software industry does not always provide satisfactory solutions to fill the gap between high-level modeling languages and programming languages. In particular, generating code from state machine diagrams constitutes one of the most challenging tasks due to their dynamic nature and because many of the state machine specification concepts are not directly supported by the object-oriented programming languages [6–11] (such as events [8], states [8,7], history pseudostates [8], or fork pseudostates [7]).

Therefore, it is not surprising that there are a wide number of proposals in the literature devoted to studying the implementation of state machine specifications in different programming languages in a wide variety of application contexts such as distributed control systems [12], decentralization of production control systems [13,14], e-voting systems [15–17], or even NASA space missions [18–20]. This number could grow in the foreseeable future because, if the Model-Driven Development (MDD) approach [1,2] overcomes its challenges [21], it will likely be increasingly applied to the development of software systems.

Taking this into account, the purpose of this paper is to review the research done in the field of object oriented code generation from state machine specifications, and provide an exhaustive analysis and comparison of these proposals.

Keeping this objective in mind, the scope of this review is limited to the literature that (i) provides proposals for generating code structures from state machine specifications, (ii) discusses the implementation possibilities of state machine specifications, and/or (iii) tackles the implementation of dynamic behaviour in general. Furthermore, in this review we take into account proposals applied generaly without specifying a certain domain for application as well as those presented for use in specific contexts. Our target readership is mainly researchers and software developers who need a satisfactory solution for implementation of the behavioural models representing the dynamics of the system they are considering. Since the needs can be extremely varied, we show all the proposals we have found trying to bring out the strengths and weaknesses of each one, thereby facilitating the selection of the most suitable solution.

In this paper we address the following research questions:

RQ1 What techniques or implementation methods have been used for generating object oriented code from state machine specifications (including UML state machines, finite state machines and Harel statecharts)?
RQ2 What state machine specification elements are translated into code by each technique or implementation method?
RQ3 What are the desired software development features considered by the published proposals? This question can also lead to another question: What approaches provide a software tool that implements the proposed methodology/technique?

The review of the research work in the field has been performed systematically carrying out a three-based dimension comparison that in turn provides an answer to each of the given research questions. On one hand, and taking into account question `RQ1`, we have analysed the general characteristics of each proposal in terms of the implementation technique it proposes. In particular, given the fact that most of the proposals are based on a pattern design [22] for implementing state machine specifications, we have classified these papers depending on whether they are based on a specific pattern or not. This first dimension of comparison corresponds to what we have called *pattern-based comparison*. On the other hand, considering question `RQ2`, we have analysed which state machine elements are considered by each proposal, distinguishing between core elements (such as state, transition, hierarchy, and concurrency) [5] and secondary elements (such as history, choice, or activity). We will refer to this dimension of comparison as *element-based comparison*. In addition, based on question `RQ3`, we have determined a taxonomy of features expected to be considered in software development such as maintenance, reusability, or memory needs. Based on such a taxonomy, we have carried out what we have called a *feature-based comparison*, which has allowed us to analyse and compare such varied proposals from a different perspective.

One of the main conclusions drawn from this review is that there are many papers that neither support several of the main components of the expressive richness of state machine specifications, such as hierarchy and concurrency [5], nor provide an implementation strategy that takes into account relevant qualitative aspects in software development such as maintenance, modularity, or reusability. Another conclusion is that UML state machines are the most common form of state machine specification used in code generation studies.

The paper is structured as follows: the next section describes the method of our systematic review, which includes inclusion and exclusion criteria, data sources and search strategy, paper selection, quality assessment and data extraction. Results are presented in Section 3. Section 4 discusses the results obtained from the review, limitations of the study and threats to validity. Finally, Section 5 covers some conclusions.

2. Research method

The method we have followed to review the research work done in the field of code generation from state machine specifications is a systematic review [23,24]. Generally speaking, a systematic review is a process by means of which all available research concerning a research subject of interest is assessed and interpreted. Such a review is undertaken through an accurate and reliable methodology stated in a review protocol. This protocol mainly specifies the research questions to be addressed, identification of research (such as databases to be searched, and search terms), selection process (methods to be used to identify, assemble, and assess the resources), assumptions and inferences to be made, and data synthesis.

Specifically, we started by developing a protocol for the systematic review, establishing in advance the methods to undertake it. Such a protocol specifies the research questions, inclusion and exclusion criteria, data sources and search strategy, paper selection, quality assessment, and data extraction.

In this section, we describe the different stages we have considered to undertake the review, while the next section is devoted to presenting the results of the systematic review according to our chosen framework.

2.1. Scope of the study

Following the PICO(C) template [25], we have identified the scope of the study as follows:

- *Population*: State machine specifications including UML state machines, finite state machines and Harel statecharts.
- *Intervention*: Techniques or implementation methods used for object oriented code generation.
- *Comparison*: Different proposals for implementing code generation for elements included in finite state specification.
- *Outcome*: The completeness of the coverage of state machine elements and the quality of the generated code.
- *Context*: Methods and techniques that identify the mechanism by which individual finite state specification elements are translated into code, excluding the generation of skeleton code such as that found in case tools.

2.2. Inclusion and exclusion criteria

In order to ensure that the studies included in the review were clearly related to the research topic, we defined detailed general guidelines for inclusion and exclusion criteria. More specifically, the main criterion used for including a paper in our review was that the study should describe quality research in the field of the implementation of state machine specifications (UML state machines, also known as UML statecharts, finite state machines and Harel statecharts). Taking this into account, only studies that: (i) give proposals for generating code from state machine specifications, and/or (ii) discuss implementation possibilities of state machine specifications, and/or (iii) tackle the implementation of dynamic behaviour in general were included in our review. We did not impose any restrictions on a specific context of application or on a particular programming language. Moreover, the systematic review included research studies published up to and including the first half of 2010.

On the other hand, we excluded pure discussion or opinion papers, and only studies reported in English were considered in the review.

A special comment must be made regarding CASE tools appearing in the marketplace with support for model to code transformation and, in particular, for code generation from state machine specifications (such as Poseidon [26], Rhapsody [27], Borland Together [28], UniMod tool [29], Rational Rose [30], and ArgoUML [31,32]). A general conclusion to be drawn from these existing CASE tools is that many of them generate limited skeleton code [33,34], not providing code generation for object behaviour, and therefore producing incomplete code. On the other hand, they commonly produce complex and cumbersome codes, making it difficult to draw any conclusions on the state-machine generated code structure. These issues make it very cumbersome to extract general and concrete rules for state machine specifications-to-code transformation. For these reasons, we have decided not to include CASE tools in our review.

2.3. Data sources and search strategy

Regarding the data sources, the search strategy for the review included electronic databases and a search of three conference proceedings. The electronic databases considered most relevant were the following: IEEE Xplore, the ACM Digital Library, ScienceDirect, Microsoft Academic Search, and Google Scholar. In addition to the electronic databases, we hand searched several volumes of the following thematic conference proceedings: ECOOP (European Conference on Object-Oriented Programming), MODELS (International Conference on Model Driven Engineering Languages and Systems), and ER (International Conference on Conceptual Modeling).

We started our systematic electronic search by identifying a list of keywords and search terms following a three-steps process (see Table 1). In the first step, in order to be as unbiased as possible, we selected a set of general keywords related to the implementation of statecharts so as to identify as many relevant papers as possible. The keywords are classified into two different groups: State machine specification and Software engineering concepts (see first step in Table 1). As a second step, in order to cover state machine specifications in general, we augmented this list with associated terms, synonyms, abbreviations, and alternative spellings, obtaining a more complete list (see second step in Table 1). With these keywords, we conducted a trial search that identified three more keywords included in different implementation proposals unknown to us at the beginning of the review (see third step in Table 1).

Fig. 1 shows the different stages performed during the systematic review process together with the number of papers identified at each stage. In Stage 1, we conducted the electronic search looking for all possible permutations of the established State machine specification and Software engineering concepts up to the moment. More specifically, we established an abstract search string constructed by separately connecting the concepts of each group (State machine specification and Software engineering) with Boolean ORs and joining the two resulting search strings with a Boolean AND. Appendix A provides a detailed discussion of the application of this abstract search string in the different database engines. It is worth noting that the abstract string was then mapped onto the different search database engines taking into account not only the range of interface forms that were provided by these databases, but also their variations in search constraints and syntax. We encountered several problems in conducting searches using the electronic databases of publications since the string needed to be separately adapted to suit the specific requirements of the electronic databases.

Regarding the search location in papers, the titles, abstracts, and keywords of articles were searched, with the exception of Google

Table 1
Keywords used in our search.

Step	State machine specification concepts	Software engineering concepts
1	UML Statecharts	Implementation
	Harel's Statecharts	Code generation
	Reactive system	Programming language
2	UML State Machines	Object-oriented language
	State charts	OO language
	Finite State Machines	C++, C#, C/C++, Java
	FSM	State-Oriented programming
	State-based behaviour	Object-oriented method
	State-based behaviour	OO method
	Dynamic models	Model-based generation
	Dynamic behaviour	Mapping
	Dynamic behaviour	Executable specification
3		Behavioural patterns
		Behavioural patterns
		Design patterns

Scholar (searching the paper titles) and Microsoft Academic Search (searching the entire paper). In addition, we performed the searches choosing the default option *All available years*. Finally, the research found in these electronic databases yielded a total of 3614 results.

For the hand search of the thematic conference proceedings (ECOOP, MODELS, and ER), we considered the publications of all conference proceedings from their beginnings to the search date (October 2010); that is, 24 editions of ECOOP (editions from 1987 to 2010), 13 editions of MODELS (editions from 1997 to 2009), and 28 editions of the ER conference (editions from 1979 to 2009). Those searches yielded a total of 9 results (3, 5, and 1 results respectively). Thus, this Stage 1 of the selection process yielded a total of 3623 papers (see Fig. 1).

The search was performed in early October 2010 and completed by the middle of the month approximately, which means that publications in the first half of 2010 are included, but some studies in the second half of the year might not have been indexed in the databases. This initial stage of the selection process was undertaken by the second author, consulting the rest of the authors when necessary, especially when looking for keyword synonyms and for agreement regarding the specification of search strings and syntax.

2.4. Paper selection

Starting from the initial list of 3623 papers, in Stage 2 the second author mainly ruled out duplicates and excluded some results with titles clearly not related to the research focus. This identification process yielded 125 papers including works published in journals, conferences, technical reports, white papers, as well as PhD Thesis and books (see Fig. 1).

In Stage 3, the second author went through the titles, abstracts, and keywords of all the studies resulting from Stage 2. In most instances, these portions provided clear indicators of whether the study adhered or not to the inclusion criteria (especially in the case of abstracts). Nevertheless, these elements did not always provide enough evidence and even some abstracts were poor and misleading. These papers, in which the inclusion/exclusion criteria were unclear, were analysed with the rest of the authors in consensus meetings, resolving disagreements by discussing the matter until agreement was reached. As a result of this stage, a total of 59 papers were chosen, which formed the basis for the next stage in our selection process.

Studies resulting from this stage were recorded in a table in a Microsoft Word document, noting for each study: (1) a unique identifier, (2) its bibliographic citation, (3) the publisher, database, or resource from which the study was available, and (4) relevant comments, if any, given in consensus by all the authors (such as the decision to include it and first impressions regarding the quality of the study).

2.5. Quality assessment

We initiated Stage 4 of the search to have a more representative set of primary studies by reading all the papers resulting from the list of papers remaining after Stage 3. In this stage, the four authors (in consensus meetings) eliminated any paper not fulfilling the criteria of papers for inclusion. More specifically, during this stage we analysed the selected 59 papers to exclude: (i) those which, despite providing a proposal for the generation of executable code from state machines, do not give enough details of how state machine elements are represented in the code generated; and (ii) those that only present simple reviews of existing proposals, applying them to particular case studies (such as tutorials).

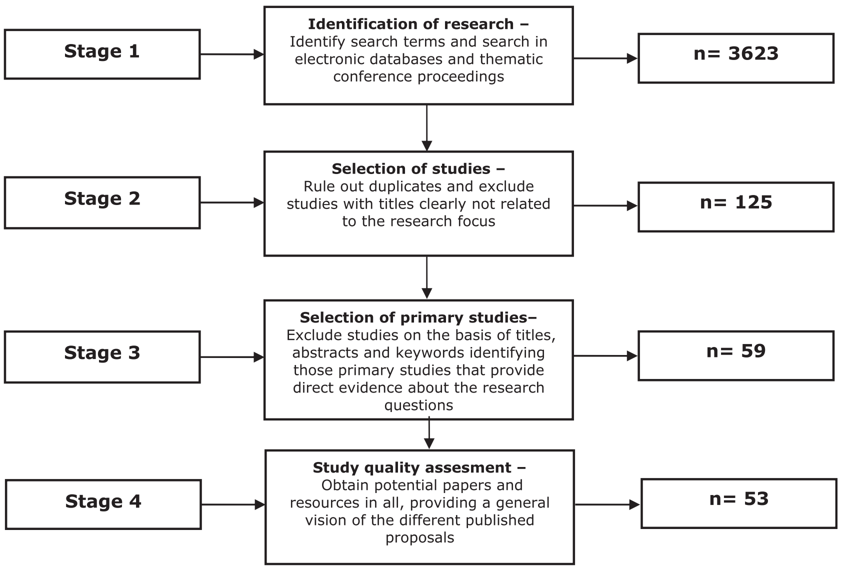
Stage 1
Identification of research – Identify search terms and search in electronic databases and thematic conference proceedings
n= 3623
Stage 2
Selection of studies – Rule out duplicates and exclude studies with titles clearly not related to the research focus
n= 125
Stage 3
Selection of primary studies– Exclude studies on the basis of titles, abstracts and keywords identifying those primary studies that provide direct evidence about the research questions
n= 59
Stage 4
Study quality assesment – Obtain potential papers and resources in all, providing a general vision of the different published proposals
n= 53

Fig. 1. Stages of the selection process.

In addition, we also consider a minimal element of quality assessment, so that a paper is included in the review if it has been published in a refereed source, or is a technical report, PhD Thesis, or book verifying that there are other papers describing the same work by the same authors published in refereed sources. The reason to consider technical reports, PhD Theses, or books relevant for our review is that normally they include more information regarding the authors' proposal for the implementation of state machine specifications, strengths, evaluation aspects, and so on than those considered in papers published by the authors in refereed sources.

Furthermore, we scanned the reference lists of all the primary studies to identify further papers; in particular, we undertook a process of snowballing, paying special attention to "Related Work" sections and reviewing papers that analyse published implementation proposals of state machine specifications. However, during this snowballing process we did not find any additional papers conforming to the inclusion criteria.

Finally, the selection process yielded 53 potential papers and resources in all, providing a general vision of the various published proposals. Moreover, this fourth stage gave us an initial notion of the strategies published in the literature regarding the research focus.

2.6. Data extraction

In this section, we explain the strategy followed to extract data from the papers, as well as the decisions made to determine the type of analysis and comparison to be performed.

We have focused on analysing the specific implementation proposals included in the 53 selected papers. In particular, taking the research questions described in Section 1, we decided to compare the different implementation proposals according to three different dimensions. On one hand, taking into account RQ1, we analysed the technique or implementation method given by each approach focusing mainly on whether the proposal is based on a specific pattern design [22] or not. We have called this dimension a *pattern-based comparison*. On the other hand, considering RQ2, we decided to compare the papers in terms of the specific state machine elements considered by each proposal. At the same time, the strategy to translate each element into code has also been analysed. We refer to this type of comparison as an *element-based comparison*. Regarding this dimension of comparison, it is worth mentioning that, although some of the analysed studies explicitly indicate the proposal's domain of application, the vast majority present a general implementation proposal without specifying a concrete domain. Therefore, in the former cases, we have had to abstract the given implementation proposal from specific issues entailed by the specific context of application. Finally, based on RQ3, we have determined a taxonomy that helps to analyse and compare these diverse proposals from a different perspective. In particular, the taxonomy has been defined on the basis of features such as readability, maintenance, or flexibility. We have called this type a *feature-based comparison*.

In order to carry out these comparisons, the authors established in consensus meetings several data extraction forms to ensure consistent and accurate extraction of the relevant information from each paper related to each dimension. In developing the data-extraction process, we determined that some of the information was needed regardless of the dimension of comparison, whereas other information was specific to each dimension of comparison. Table 2 shows the data extracted from all papers regardless of the dimension of comparison.

In contrast, Table 3 shows the data that were extracted, when possible, from all selected papers regarding the *pattern-based* comparison. In addition to identifying the implementation strategy of each proposal, we have identified related work presented in the papers in order to gather additional information regarding the strengths and weaknesses of other proposals. As concerns the *elements-based* comparison, we have obtained from each proposal the implementation approach of state machine elements as regards: context-class, current state, simple states, composite states (both simple and orthogonal), transition elements (state transition process, event, guard, and action), pseudostates (fork, join, choice, shallow history, and deep history), and activities. Finally, Table 4 includes the data extracted from all the selected papers to carry out the *feature-based* comparison strategy. In particular, these data items, classified into four groups, have been evaluated in accordance with the following criteria: (i) 'B/M/G' represents *Bad*, *Medium*, or *Good*; (ii) 'NS/PS/S' refers to *Not Supported*, *Partially Supported*, or *Supported*, (iii) 'N/Ne/Y' refers to *No*, *Neutral*, or *Yes*; and (iv) 'L/M/H' represents *Low*, *Medium*, or *High*.

The way in which certain papers were reported made it difficult to extract some of the data items called for in the forms (especially data items related to the *element-based* and *feature-based* comparison). Therefore, every paper included was read in detail and the data was extracted and cross-checked by all the authors in consensus meetings.

Table 2
Data items extracted from all papers.

Data items	Description
Reference	Unique identifier for the paper (same as the bibliographic reference number)
Bibliographic	Author (s), year, title, source, web-site (if possible)
Type of article	Journal/conference/PhD thesis/books/technical report/ white paper
Paper goal	The goal of the paper
Related papers by authors	References of related works by the same authors

Table 3
Data extracted from each study regarding the pattern-based comparison.

Data items	Description
Strategy	Pattern-based or not, and if so, what the design pattern is
Related papers	Other proposals the authors compare their own with

3. Results

This section describes the analysis of the data extracted from the selected papers. First, we present in Section 3.1 the main characteristics of the 53 studies in the review and then, in Sections 3.2, 3.3, and 3.4, we synthesize the data from all the papers to answer each question as described in Section 2.6. A complete explanation of certain proposals presented herein regarding the three comparison aspects is presented in [35].

3.1. Presentation of the studies

We have classified the selected references into 28 different studies as identified in Table 5, where we have considered for each study: (i) a unique identifier used from now onto refer to each study (*Identifier*), (ii) a representative authors' name of all papers published by the same authors describing similar research work (*Authors' R.N.*), and (iii) the bibliographic references of the papers (*References*).

Taking into account the years of publication of the first and last studies included in the review, the selected papers cover the timespan 1992–2010.

Table 4
Data extracted from each study regarding the feature-based comparison.

Type	Data items	Description
State machines' expressivity	Hierarchy	Whether the study gives a proposal for the implementation of states' hierarchy (NS/PS/S)
	Concurrency	Whether the study supports the implementation of states' concurrency (NS/PS/S)
	History	Whether the study provides a proposal for history pseudostates (NS/PS/S)
Software design	Expandability	The degree to which the design of a system can be extended (B/M/G)
	Simplicity	The degree to which the design of a system can be understood easily (N/Ne/Y)
	Reusability	The degree to which a piece of design can be reused in another design (B/M/G)
Implementation	Learnability	The degree to which the code source of a system is easy to learn (B/M/G)
	Understandability	The degree to which the code source can be understood easily (B/M/G)
	Modularity	The degree to which the implementation of the functions of a system are independent from one another (B/M/G)
Runtime	Performance of execution	How well the generated system executes (B/M/G)
	Memory needs	The amount of computer memory needed by the system's execution (L/M/H)
	Efficiency	How well the system utilizes processor capacity, disc space and memory (B/M/G)
Final result	Tooling	Corresponds to whether the authors provide a tool implementing their proposal

Table 5
Selected studies and associated references.

Identifier	Authors' R.N.	References
S1	Adamczyk	[36,37]
S2	Ali	[11,38]
S3	Babitsky	[39]
S4	Benowitz et al.	[18–20]
S5	Chauvel et al.	[40]
S6	Chow et al.	[34]
S7	Culwin	[41]
S8	Derezinska et al.	[42–45]
S9	Douglass	[46,47]
S10	Duby	[48,49]
S11	Gamma et al.	[22]
S12	Gurp et al.	[4]
S13	Heinzmann	[50]
S14	Knapp et al.	[51,52]
S15	Köhler et al.	[13,14]
S16	Lafreniere	[53]
S17	Lamour et al.	[54,55]
S18	Metz et al.	[56]
S19	Pintér et al.	[57–61]
S20	Samek	[62]
S21	Samek et al.	[63,64]
S22	Saúde et al.	[65]
S23	Shalyto et al.	[66]
S24	Shlaer et al.	[67]
S25	Tanaka et al.	[6–8,68–70]
S26	Tiella et al.	[15–17]
S27	Tomura et al.	[12,71]
S28	Yacoub et al.	[72]

Looking at the number of papers by year of publication in Fig. 2, we notice an increasing interest in the area from 1995 onwards. We also note a slight decrease in the number of papers published since 2004, but papers published in recent years show that the problem is still unresolved.

Regarding the context of application, although some of the analysed studies explicitly indicate the application domain (distributed control systems [S27], decentralization of production control systems [S15], fault-tolerant systems [S17], e-voting systems [S26], or fault protection subsystem of a space mission [S4]), the vast majority of the works present their implementation approach generally, without specifying a concrete domain where it could be applied.

Taking into account the data extraction framework outlined in Section 2.6, the characteristics of these 53 selected papers classified into the 28 studies are given in Tables 6 and 7. Table 6 presents studies that provide reviews of other works or apply existing proposals from other authors. Table 7 shows studies that provide implementation proposals themselves. In these tables we have distinguished: (i) the identifier of each study, (ii) the bibliographic reference of the papers selected for inclusion in the review (*Ref.*), (iii) the source of publication of each of these papers (*Source*), (iv) the year of publication of each paper (*Year*), (v) the goal of the work (*Goal*) distinguishing between 'GP' (if the work includes a state machines' implementation approach within a General Project/approach) and 'PIP' (if the work just Provides an Implementation Approach), (vi) the behavioural model the authors consider to be implemented (*Behavioural model*), and (vii) the object-oriented programming language used to implement the authors' approach. Where the authors have more than one paper related to the given issue, we have taken as the authors' reference name either the surname of the first author signing the oldest published paper or the surname of the author who appears in most of the cited papers.

Specially, as presented in Tables 6 and 7, most of the selected studies (20 studies) focus on providing a state machine implementation approach, whereas a few (eight studies) form part of more general projects or works that do not have this implementation as the only goal. From these tables we can also infer that most of the studies (19 studies) consider UML Statecharts as behavioural model to be implemented. Another conclusion is that some authors use different object-oriented programming languages to implement their approaches (for example, 13 of the studies use Java, 10 use C/C++, and 2 use C#), and four studies do not provide this information or simply leave the actual decision up to the user, which has been represented in Tables 6 and 7 with the acronym 'UTU'.

In particular, the 53 selected works with the 28 studies can be classified into three different groups as described in the paragraphs below.

First, we highlight the study [S1] by Adamczyk (see Table 6) in which the author presents an expert review comparing more than twenty possible implementations and extensions of the *State pattern* (which, as we will see below, is one of the most commonly used implementation proposals). The comparison focuses mainly on presenting the advantages of each pattern but not discussing its drawbacks sufficiently. Thus, the author presents, for each pattern, the applicability context, a solution to tackle the context, consequences of the design proposal, and related patterns. The two papers included in study [S1] have been used to clarify ideas of implementation proposals appearing in other works, as well as to compare them regarding their applicability, advantages, and disadvantages.

Second, we have considered another four different studies ([S4], [S5], [S22], and [S26]), including eight different papers (see

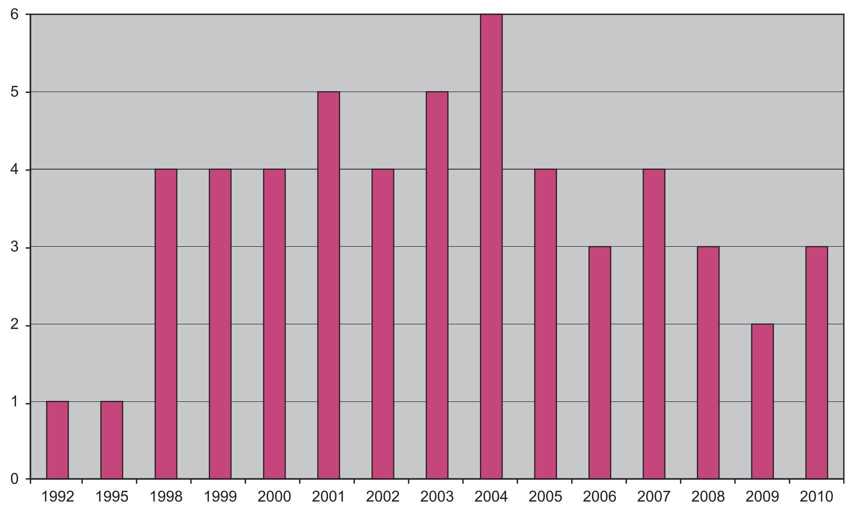

Fig. 2. Publications by year.

Table 6
General aspects of studies making use of other works.

Id.	Ref.	Source	Year	Goal	Behavioural model	Pro. lang.
S1	[36]	Proceedings	2003	PIP	Finite State Machines	Java/C/C++
	[37]	Proceedings	2004			
S4	[18]	Proceedings	2006	GP	UML Statecharts	C/C++
	[19]	Proceedings	2008			
	[20]	Tech. Rep.	2008			
S5	[40]	Proceedings	2005	PIP	UML Statecharts	Java
S22	[65]	Proceedings	2010	PIP	Finite State Machines	Java
S26	[15]	Proceedings	2006	GP	UML Statecharts	Java
	[16]	Proceedings	2007			
	[17]	Journal	2009			

Table 6), not because they provide new ways of implementing state machine specifications, but because they apply existing implementation approaches from other authors to their specific works. We decided to include them since they normally highlight the main characteristics of the implementation proposal used, even comparing it to existing approaches, which may by particularly useful for the review. In particular, the study [S4] by Benowitz et al. uses the *QHsm pattern* presented in [62], study [S26] applies the *State pattern* approach given in [22], and [S22] integrates the *State Machine Design pattern* presented in [66].

We have also included within this group the study [S5] by F. Chauvel and J. Jézéquel, which tackles the implementation of UML statecharts in general, providing an upper-level general framework that can be adapted to any existing state machine implementation proposal. More specifically, first, the authors propose to reify the various *semantic variation points* of UML 2.0 statecharts [3] into models of their own in order to avoid hardcoding the semantic choices in the tools. Second, the authors propose modelling several implementation choices in the same spirit as in the modelling of semantic variation points. Finally, following an OMG's Model Driven Architecture, these semantic and implementation models are processed along with a source UML model used as a basis for code generation.

Finally, in contrast to the previous studies, the remaining 39 papers themselves provide 23 different implementation proposals for generating code from state machine specifications (see Table 7). Each of these studies provides support for the implementation of a specific subset of state machine elements, providing most of them with different frameworks that include, to a greater or lesser extent, desired software development features. Furthermore, these papers usually have a related work section in which the authors synthesize related literature, which is an interesting component to help us to assess and evaluate the considered implementation proposals.

Therefore, taking into account the research questions, all of the 28 studies have been considered for the review as follows. On one hand, the first five studies (Table 6) have been included in the review mainly to clarify ideas and to be used as a preliminary validation of the implementation proposals appearing in the other 23 studies since these five studies are expected to highlight their strengths and weaknesses. On the other hand, the remaining 23 studies (Table 7), which provide their own state machine implementation proposals, have been taken as sources for the different comparisons we perform in the following sections, using as source background the above five studies.

3.2. Pattern-based comparison (RQ1)

The main goal of this comparison is to distinguish the different methodologies proposed for the selected studies regarding the implementation of state machine specifications. Next, we introduce these proposals giving a brief explanation of each of them, starting with the pattern-based proposals and continuing with those not based on any pattern. In order to provide the reader with a general view of the different proposals, a summary is presented in Table 8, for pattern-based proposals, and Table 9 for proposals not based on any pattern. These two tables employ the following notation: (i) an empty cell represents that the authors do not mention anything about the implementation proposal in question, (ii) the symbol 'IB' indicates that the authors' approach Is Based on the corresponding implementation proposal. Moreover, we have included additional information when the authors compare their works with the implementation proposal shown in the corresponding row.

Regarding the strategy followed by the authors, we would like to point out that in Tables 8 and 9 we have mainly distinguished two different options. (1) Whether the authors present a specific pattern, in which case we have noted the pattern, or if they extend an existing pattern (such as the proposal given by Heinzmann in [S13], which is an extension of the *Hierarchical State Machine pat-*

Table 7
General aspects of studies providing implementation proposals.

Id.	Ref.	Source	Year	Goal	Behavioural model	Pro. lang.
S2	[11]	Journal	2010	PIP	UML State Machines	Java
	[38]	Proceedings	2010			
S3	[39]	Journal	2005	PIP	UML Statechart Diag.	C/C++
S6	[34]	Proceedings	2000	PIP	UML Statechart Diag.	Java
S7	[41]	Journal	2004	PIP	UML Statecharts	Java
S8	[42]	Proceedings	2007	GP	UML State Machines	C#
	[44]	Journal	2007			
	[43]	Proceedings	2008			
	[45]	Journal	2009			
S9	[46]	Book	1998	PIP	UML Statecharts	C/C++
	[47]	Proceedings	2001			
S10	[48]	Proceedings	2001	PIP	UML Statechart Diag.	C/C++
	[49]	White paper	2004			
S11	[22]	Book	1995	GP	–	UTU
S12	[4]	Proceedings	1999	PIP	Finite State Machines	Java
S13	[50]	Journal	2004	GP	Hierarchical State Mach./ UML Statecharts	C/C++
S14	[51]	Proceedings	2002	GP	UML State Machines	Java
	[52]	Proceedings	2002			
S15	[13]	Tech. Rep.	1999	GP	UML Statecharts	Java
	[14]	Proceedings	2000			
S16	[53]	Journal	2000	PIP	UML State Machines	C/C++
S17	[54]	Proceedings	1998	PIP	Harel's Statecharts	Java
	[55]	Proceedings	1998			
S18	[56]	Proceedings	1999	PIP	UML Statechart Diag.	C/C++
S19	[57]	Journal	2003	PIP	UML Statecharts	UTU
	[61]	Proceedings	2003			
	[58]	Proceedings	2003			
	[59]	Proceedings	2004			
	[60]	PhD Disser.	2007			
S20	[62]	Book	2002	PIP	UML Statecharts	C/C++
S21	[63]	Journal	2000	PIP	UML Statecharts	C/C++
	[64]	Book	2002			
S23	[66]	Proceedings	2006	PIP	State charts	C#
S24	[67]	Book	1992	PIP	Harel's Statecharts	UTU
S25	[68]	Proceedings	1999	PIP	UML Statecharts	Java
	[69]	Journal	2001			
	[6]	Proceedings	2003			
	[7]	Proceedings	2004			
	[8]	PhD Disser.	2005			
	[70]	Journal	2005			
S27	[12]	Proceedings	2001	GP	Finite State Machines	Java
	[71]	Proceedings	2001			
S28	[72]	Proceedings	1998	PIP	Finite State Machines	UTU

tern proposed by [S21]). Or (2) whether the authors do not use any pattern as a base, in which case we consider another two options: (2.a) the proposal is based on a known approach (such as the use of *switch statements* by Metz et al. in [S18]), or (2.b) the proposal is a concrete one, not based specifically on any existing approach (such as the one given by Knapp et al. in [S14]), in which case we note the authors' surnames. More specifically, from the 23 different implementation proposals, 15 correspond to pattern-based approaches and the other 8 do not.

3.2.1. Pattern-based proposals

A design pattern [22] is not expected to describe the details of the implementation since it only specifies a general solution for recurring design problems [56]. Consequently, when considering pattern-based proposals, it must be taken into account that, although they are based on the implementation guidelines provided by the pattern, the actual implementation decisions were made by the developers. In particular, regarding studies based on a specific design pattern, in this review we discuss the most commonly used patterns and also include those works that have proposed extensions to some such patterns, providing an approach for the implementation of particular features not supported by the source pattern-based proposal.

The pattern-based comparison regarding pattern-based proposals is summarized in Table 9, showing, as mentioned above, the implementation technique followed in each work, whether the work corresponds to an extension of another proposal, and the studies the authors compare their own with. Next we describe the main characteristics of these proposals.

3.2.1.1. State pattern. The *State pattern*, or *State Design pattern*, was first introduced by Gamma et al. in the study [S11] together with a total of 23 useful patterns for systems design. This pattern is considered to be a useful software pattern that takes advantage of *polymorphism* to define different behaviours for different states of an object. It is especially valuable to master because it can be used in practically any size application [32]. As mentioned above, [S26] applies this pattern to their specific context of application.

In addition, the *State pattern* (as well as the *State Table pattern* corresponding to study [S9] that we present below) primarily focuses on encapsulating only the behaviour of the state of the context object, which is problematic when dealing with specific behaviours and substates. In particular, this pattern does not deal with the hierarchical states, history or concurrency, some of the most commonly used elements of state machine specifications [8,63,57], which are unsupported. Furthermore, another important weakness of this approach is that it does not provide any means of implementing the model's dynamic parts [8,57].

Given the pattern weaknesses, other proposals have been built upon it, providing some solutions. The proposals advanced by Tanaka et al. in [S25] is an extension of the *State design pattern* and mainly solves the aforementioned substate problems.

3.2.1.2. Finite State Machines (FSMs) framework. Gurp et al. [S12] presented the Finite State Machines (FSMs) framework to implement Finite state machines as an alternative approach to solving certain problems related to the evolution of FSM implementation and data management. In particular, in their work Gurp et al. examine two of the most commonly used implementation approaches: doubly nested switch statements (or procedural approach) and the *State pattern*. They discuss the main problems that arise through the use of such proposals and present a solution that addresses the problems providing more structure at the implementation level by modelling all the state machine elements as classes.

3.2.1.3. State-Table pattern. Another pattern used for UML statechart implementation is the *State-Table pattern* first defined by Douglas [S9]. There are several approaches for this pattern, but the most commonly used one considers the state table consisting of an $n \times m$ array, where n is the number of states and m is the number of transitions, and each cell (row/column intersections) contains a single pointer to a transition object that handles the event [46]. Nevertheless, the exact data structuring depends on the problem being solved [47].

Table 8
Pattern-based comparison for pattern-based proposals.

Approach	References	Gamma et al. S11	Tanaka et al. S25	Gurp et al. S12	Douglass S9	Duby S10	Köhler et al. S15	Yacoub et al. S28	Shalyto et al. S23	Tomura et al. S27	Samek et al. S21	Heinzmann S13	Samek S20	Babitsky S3	Pintér et al. S19	Lamour et al. S17
Pattern-based	State pattern	IB	IB	Maintenance limitations, FSM instantiation and data management problems			Complicated, large inheritance hierarchy of many specific, small state classes that mainly redefine a small number of methods	No support for orthogonal, concurrent states, and history states	Too verbose, the implementation of states in different classes causes distribution of the transition logic among these classes	Critical problems with support of substates, mapping from diagram to code, and readability of code	Performance relatively poorly if the state machine is complex		Makes state transition efficient, good performance for event dispatching. Memory efficient, no support for hierarchy	IB	Elegant and efficient separation of concerns, does not provide any means for implementing the dynamic parts of the model, no explicit support for hierarchy, history and concurrency.	Complex implementation of control aspects in complex behaviour
	FSM Framework		State trans. is about twice as expensive as in State pattern for simple transition. Not optimized code. Time events and history states are not implemented	IB												
	State-Table pattern		Has critical problems with suppport for substates, mapping from diagram to code, and code readability		IB	IB	IB			Critical problems with support of substates, mapping from diagram to code, and readability of code	Cumbersome, requires a large action/state array and many fine-granularity functions representing actions		Popular, provides relatively good performance, promotes code reuse but requires large state table, a manual maintenance of this initialization is expensive and prone to error, no support for hierarchy	No comparison provided	Most of the large pointer tables are unused, slightly faster than the nested switch structure but requires much more memory, no support for hierarchy, history, concurrency, etc. and the resulting code is inflexible	
	Basic Statechart pattern		Has critical problems with suppport for substates, mapping from diagram to code, and code readability					IB								
	State Machine Design pattern								IB							
	Statechart Design pattern									IB						
	Hierarchical State Machine pattern		Has critical problems with suppport for substates, mapping from diagram to code, and code readability								IB	IB (an extension)				
	Quantum Hierarchical State Machine												IB	IB (an extension)	IB (an extension)	
	Reflective State pattern															IB
Other	Switch statements		It has critical problems with suppport for substates, mapping from diagram to code, and code readability	Duplicated code, redundant code, difficulties to reuse code, tedious maintenance (bugs have to be fixed more than once, easy to forget a piece of code, complex code)						Critical problems with support of substates, mapping from diagram to code, and readability of code.	Works well for classic "flat" SM and is widely used by automatic code synthesizing tools. Manual coding of entry/exit actions and nested states is cumbersome, difficult to modify and maintain.		Very popular and simple but has code reuse problems, no support for hierarchy, manually coded entry/exit actions, and nested initial transitions are prone to error and difficult to maintain.	No comparison provided	Code repetition, maintenance is hard, error-prone, and labour intensive, Does not provide explicit means for reflecting the transition structure, state hierarchy and entry/exit actions.	
	Knapp et al.		Does not produce optimized code. Time events and history states are not implemented													
	Shlaer et al.		Hierarchy, concurrency, entry/exit and transition actions not supported													
	Culwin															
	Chow et al.															
	Derezinska et al.															
	Lafreniere															
	Ali															

Table 9
Pattern-based comparison for proposals not based on any pattern.

	References / Approach	Metz et al. S18	Knapp et al. S14	Shlaer et al. S24	Culwin S7	Chow et al. S6	Derezinska et al. S8	Lafreniere S16	Ali S2
Pattern-based	State pattern	Simple metamodel and understandable object interaction, high memory resources, code redundancy					Explosion of number of classes		Does not support state hierarchy and concurency
	FSM Framework								No comparison provided
	State-Table pattern	No code redundancy but complex model							No comparison provided
	Basic Statechart pattern								
	State Machine Design Pattern								
	Statechart Design pattern								
	Hierarchical State Machine pattern								
	Quantum Hierarchical State Machine								
	Reflective State pattern								
Other	Switch statements	IB						No way to enforce the state transition rules, rarely suitable for use in a multithreaded system	Earliest technique that works well for classic flat state machines and is mostly used in non-object-oriented procedural languages.
	Knapp et al.		IB						No comparison provided
	Shlaer et al.	No code redundancy, conceptually well-founded but very complicated, need to allocate additional space for their transition instances		IB					
	Culwin				IB				
	Chow et al.					IB			
	Derezinska et al.						IB		
	Lafreniere							IB	
	Ali								IB

Other proposals, such as the one given by Duby [S10] and the approach given by Köhler et al. [S15], have extended this approach in different ways. Duby [S10], for instance, advocates the *state table* technique [S9] based on pointer-to-member functions in C++, considering a reactive system to be composed of a set of `ActiveObjects` instances that respond to events. On the other hand, Köhler et al. [S15] adapt the idea of Douglas, providing an object-oriented implementation of the state-table at runtime.

3.2.1.4. Basic Statechart pattern. Yacoub et al. [S28] propose several patterns of Finite State Machines. Particularly, they define the *Basic Statechart pattern* which is an extension of the *State pattern* to implement guards and entry/exit actions. So, support for hierarchy, concurrency, and history remains a pending task. Additionally, Yacoub et al. [S28] propose other patterns, however, in this review, we only take into account the *Basic Statechart pattern* because it provides the basis concepts and implementation philosophy of their other patterns.

3.2.1.5. State Machine Design pattern. Shalyto et al. [S23] present this pattern as an extension of the *State pattern* with the aim of making the classes designed with the *State Machine pattern* more reusable than the ones designed with the *State pattern*. To do so, the authors introduce an event mechanism. Specifically, the basic idea of the *State Machine* pattern is to separate classes that implement transition logic (*context* class) and state classes so that the *context* and *state* classes interact by using events as objects that the *state* objects pass to *context*.

The proposal by Saúde et al. [S22] shows the feasibility of using this pattern with large databases, and equally large business processes and workflows. In particular, the authors present the *Persistent State pattern* which integrates classic and enterprise design patterns (*Abstract Factory design pattern* [22], *Data Access Object design pattern* [73], and *Business Object pattern* [73]), together with the *State Machine Design pattern*, which is specifically used to control the transition logic of the state machine description.

3.2.1.6. Statechart Design pattern. Tomura et al.'s work [S27] proposes the *Statechart design pattern* for Finite State Machines. This pattern defines classes and state-transition execution mechanisms for realizing the dynamic behaviour of device component models of an open distributed control system.

3.2.1.7. Hierarchical State Machine pattern. The *Hierarchical State Machine pattern* (HSM), by Samek et al. [S21], as well as the *Quantum Hierarchical State Machine pattern* (QHsm) by Samek [S20] presented below, have been defined to implement state hierarchy and transition dynamics. In particular, the fundamental HSM pattern constitutes an external add-onto C++ or C. Unlike the *State pattern*, the `State` *class is not intended for subclassing but rather for inclusion. Therefore, in their proposal, state machines are defined by composition rather than by inheritance.*

Heinzmann [S13] provides an extension of the *Hierarchical State Machine pattern* [S21]. In particular, this approach allows the codification of a state machine in C++ directly, without the need for special code generation tools or wizards, although the author acknowledges that the existence of such a wizard could facilitate further development by converting the statechart automatically. Heinzmann uses template techniques to enable the compiler to optimize the code through inlining.

3.2.1.8. Quantum Hierarchical State Machine pattern (QHsm). This pattern, proposed by Samek [S20], is an improved version of HSM that takes advantage of the Quantum Programming paradigm (QP). QP enables statechart modelling directly in C or C++ through two further fundamental meta-patterns: the HSM and active-object based framework.

Study [S4] uses the Quantum framework building rapid executable and verifiable models that can be implemented directly into the application software. Furthermore, two proposals for extensions to this pattern have been provided by Babitsky [S3] and Pintér et al. [S19]. In particular, Babitsky in [S3] proposes an approach that is a follow-up to Samek's work [S20] and, at the same time, is based in part on the *State design pattern*. On the other hand, Pintér et al. [S19] provide an extension of the *Quantum Hierarchical State Machine pattern* that they call *Extended Quantum Hierarchical state machine (EQHsm)*, providing support for actions on transitions, history states, and most cases of concurrent operation (that is, concurrency is partially supported).

3.2.1.9. Reflective State pattern. Lamour et al. [S17] present this pattern as a refinement of the *State pattern* based on the *Reflection architectural pattern*. The proposed pattern solves some of the design decisions that have to be made to implement the *State pattern* (such as the creation of the control of *state* objects and also the execution of state transitions) or even some disadvantages of the *State pattern* (such as centralizing all the control aspects in the *context* object or decentralizing the responsibilities of the transition logic), by allowing *state* subclasses themselves to specify their successor states. Specifically, it uses the *Reflection architectural* pattern to separate the control aspects of the state machine's implementation from the application's logic [36]. Nevertheless, one of the disadvantages is that not all programming languages support reflection [36].

3.2.2. Proposals not based on design patterns

The results of the pattern-based comparison regarding proposals not based on any design pattern are summarized in Table 9. Next we describe the main characteristics of these proposals.

3.2.2.1. Nested switch statements. Nested switch statements [46] constitutes one of the most common proposals to implement state machine specifications. This proposal normally uses doubly nested switch statements for partitioning the event-handler function to segments reflecting the object behaviour in specific states (external branches) and sub-segments for each event handled in the state (internal branches) [46,47,57,63]. As stated in [11], this technique works well when implementing flat state machines and is mostly used in non-object-oriented procedural languages. An example of this method's use can be seen in the work presented by Metz et al. [S18], where an ordinary switch statement in C++ is used to perform the state transition process of the system.

3.2.2.2. Knapp et al.'s proposal. Knapp et al. [S14] present *Hugo* as a UML model translator for model checking (using the SPIN [74] model checker), theorem proving, and code generation. Hugo code generation is interpretative in nature and is not aimed at producing product quality and optimized code [51,75], but is intended instead to represent UML semantics as faithfully as possible. Taking into account the works published in relation to this project, it seems that the tool efforts are oriented towards verification rather than code generation. In fact, we have not been able to extract clear rules of their code generation approach due to the incomplete information presented in the paper. So, in this review we consider literally the main points of their approach we have obtained from Refs. [51,52,75].

3.2.2.3. Shlaer et al.'s proposal. Shlaer et al. [S24] suggest an implementation of a subset of Harel's statecharts [5] based on a linked list of transitions.

3.2.2.4. Culwin's proposal. Culwin [S7] gives a proposal for the implementation of a specific statechart representing a date input mechanism, without focusing on describing general implementation rules. In particular, we should note that, since the author presents her proposal by directly applying it to such a specific statechart, it has been a bit difficult to extract the general rules for statechart to Java code transformation.

3.2.2.5. Chow et al.'s proposal. Chow et al. [S6] give an approach for translating code from the dynamic behaviour of a system based on two main steps. First, in the *statechart diagram* step, the corresponding Java code is generated from each element. Second, the *generate method body* step is based on the pre/post condition of an operation and specifies the order of language statements taking into account the message-passing sequence in the interaction diagram.

3.2.2.6. Derezinska et al.'s proposal. Derezinska et al. [S8] present a framework for eXecutable UML (FXU) that is able to transform UML models into C# source code and supports execution of the application reflecting the behavioural model. The authors state that this framework corresponds to the first solution that supports generation and execution of all elements of state machine UML 2.0 [76] using the C# language. The FXU Framework consists of two components: the FXU Generator, responsible for the transformation of the UML model to the corresponding C# code realization, and the FXU Runtime Library, which implements the general rules of state machine behaviour (such as processing of events, execution of transitions, entering and exiting states, and realization of different pseudostates).

3.2.2.7. Lafreniere's proposal. Lafreniere [S16] presents a particular design for the implementation of state machines using C++, which solves the problems given by the use of switch statements by including support for both internal and external events, event data, and state transition validation.

3.2.2.8. Ali's proposal. Ali [S2] aims to provide an easy way for implementing concurrent/hierarchical state machines into efficient and encapsulated Java code based mainly on the use of Java `Enums` [77], making the resulting code compact, efficient and easy to understand [11,38]. His approach considers state machines having hierarchical and concurrent states, and encapsulates the state machine behaviour within a specific class and keeps the structure of the state machine obvious at the programming level.

3.3. Element-based comparison

The goal of this comparison is to analyse which state machine elements are considered by each proposal and the strategy followed in order to implement these elements as code structures. The results are summarized in Table 10, for pattern-based proposals, and in Table 11, for not pattern-based proposals, indicating in

Table 10
Element-based comparison of pattern-based proposals.

State machine specification elements / References		Gamma et al. S11	Tanaka et al. S25	Gurp et al. S12	Douglass S9	Duby S10	Köhler et al. S15	Yacoub et al. S28	Shalyto et al. S23	Tomura et al. S27	Samek et al. S21	Heinzmann S13	Samek S20	Babitsky S3	Pintér et al. S19	Lamour et al. S17
Context Class		Class	Class	Class	Class	Sublass	Subclass		Class	Class	Subclass	Class	Subclass	Class	Subclass	Class+ metaclass
Current State		Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute	Attribute
Simple State		SS1	SS1	SS3	SS3	SS3	SS3	SS1	SS2	SS3	SS3	SS3	SS4	SS1	SS4	SS6
State transition process		Switch Stat.	Switch Stat.	Switch Stat.	State-Table structure	State-Table structure	State-Table structure	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.
Transition	Events	E1	E1	E2	E2	E2	E2	E2	E2	E3	E4	Optional	E2	E1	E2	E1
	Guards		if-statem.	X	Evaluated method	if-statem.	Evaluated method	if-statem.		Evaluated method	--	if-statem.	if-statem.			--
	Actions	A1	A1	A2	A1	A1	A1	A3		A1		A1	A1		A1	--
Composite State	Simple	X	A class extension of ABC, a region abstract class derived classes	X	X	Transition chains	A class for all composite states + links state objects	X		Instance of a class + links to state objects	A class `State` for all states	A template class	Member functions	A class per each state	Member functions	X
	Orthogonal	X	A class extension of ABC, region abstract classes, derived classes	X	X	Transition chains	A class for all orthogonal states + links to state objects	X		Class	X	X	X	Class	Multiple SM with wrapper states + events*	X
Pseudostates	Fork		Method +													
	Join		if-statem. +													
	Choice		if-statem.										if-statem.			
	Shallow His.	X	Attribute				Attribute	X		X	Attribute	X			Attribute	
	Deep His.	X					Attribute	X		X		X	Attribute		Attribute	
Activities (entry/exit/do actions)			A3 (en/ex)	A4 (en/ex/do)	A1 (en/ex/do)	A1 (en/ex)	A1 (en/ex)	A3 (en/ex)		A1 (en/ex/do)	A2 (en/ex)	A1 (en/ex)	A1 (en/ex)	A3 (en/ex)	A3 (en/ex)	

Table 11
Element-based comparison of not pattern-based proposals.

State machine specification elements \ References		Metz et al. S18	Knapp et al. S14	Shlaer et al. S24	Culwin S7	Chow et al. S6	Derezinska et al. S8	Lafreniere S16	Ali S2
Context Class		Class	Class	Subclass	Class	Class	Class	Subclass	Class
Current State		Attribute		Attribute	Attribute	Attribute	Attribute	Attribute	Attribute
Simple State		SS5	SS3	SS5	SS5	SS5	SS3	SS4	SS5
State transition process		Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.	Switch Stat.
Transition	Events	E1	E2 (Time events X)	E1	Dep. on the example	E1 (Time events X)	A class per each event type	E1	E4
	Guards	if-statem.	A class for all guards		if-statem.	if-statem.	Evaluated method		if-statem.
	Actions	A1	A4			A1	A1		A1
Composite State	Simple	Enumeration type	--	X	Method		--		A `State` enum reference in the `State` enum
	Orthogonal	X	--	X			--		Object composition
Pseudostates	Fork						A class for all forks		
	Join						A class for all joins		
	Choice						A class for all choices		
	Shallow His.	Attribute+ method					A class for all shadow h.		
	Deep His.						A class for all deep h.		
Activities actions	(entry/exit/do	A1 (en/ex/do)					-- (en/ex/do)		A1 (en/ex)

every cell of these tables each author's proposal. We decided to present the results using these tables in order to ease legibility, but the explanations of the results are given without making the same distinction. In these tables, we have used the following notation: (i) an empty cell means that the authors do not mention anything about the element in question, (ii) the symbol '–' indicates that the authors propose translating the corresponding element but they do not explain their complete strategy, and (iii) the symbol 'X' means that the authors explicitly state that they do not provide an implementation for the element. In all events, we have filled each cell with the authors' translation proposal for the corresponding element.

In order to compare the way in which each element is considered by each proposal, we have sorted the rows into three groups (represented by alternating grey/white colours in Tables 10 and 11), using a double criteria. The first group is determined according to the number of works that deal with each element. More specifically, this group includes the elements for which the majority of works propose a translation. In contrast, the elements of the other two groups are analysed by either few works or almost none. The second and third group distinguish between composite states, considered as core elements [5] that we believe should be considered by the authors, and pseudostates and activities, which can be thought of as secondary elements and therefore are not essential to be translated. Next, we will explain each group in detail.

As mentioned above, the *first group* includes the elements that are considered by almost every proposal. It is no wonder that the elements of this group are the most widely used state machine specification elements: *current state*, *simple state*, *event*, *guard*, *action*, considering also the *state-transition process* and the *context class*, whose behaviour is represented by the state machine specification. With regard to the strategy followed to implement the *context class*, the *current state* and the *state-transition process*, the authors are in agreement, to a greater or lesser extent. In particular, the usual proposal for the implementation of the *context class* is by means of a simple class, while others propose to also define a base class from which the previous one will inherit. The specific class, in the former case, or the corresponding superclass, in the latter case, describes the state machine behaviour.

As regards *current state*, all approaches agree in their representation as an attribute whose value refers to the current state the object is in.

Regarding *simple states*, six different strategies are distinguished:

- SS1: defining a purely abstract class for every state plus a concrete subclass, which inherits from the general class, for each operational state ([S3], [S11], [S25], and [S28]).
- SS2: creating a specific class for each state in the state machine specification ([S23]).
- SS3: defining a single class for all states ([S8], [S9], [S10], [S12], [S13], [S14], [S15], [S21], and [S27]).
- SS4: creating a function for each state in the context class ([S16]) or in derived classes ([S20] and [S19]).
- SS5: defining an `enumeration` type for all states (each state as an `enum` value) ([S2] and [S18]) or simply a data value for each state ([S6], [S7], and [S24]).
- SS6: defining several classes representing the state object initializations and the behaviour associated with each state ([S17]).

The performance of the *state-transition* process is encapsulated in a specific method (in most cases located in the context class) that is invoked for each incoming event and is responsible for delegating events to the corresponding state implementation structures or directly to the state implementation of that event. The trigger of the transition is checked normally by a switch statement; exceptions to this implementation proposal are the approaches based on the *State-table pattern* ([S9], [S10] and [S15]), in which the state transition correspondences are encapsulated in a state-table structure as described in the previous section.

Events are implemented as:

- E1: a method in the corresponding state class ([S3], [S11], [S17], and [S25]) or in the context class ([S6], [S16], [S18] and [S24]).
- E2: an object of a single class defined for all events ([S8], [S9], [S14], [S19], [S20] and [S23]), in some cases, together with a specific method implemented accordingly in the class ([S10]), in the context class ([S15]), or in the corresponding state class ([S28]).
- E3: an attribute in a transition class ([S27]).
- E4: a value in an `enumeration` type ([S2] and [S21]).

Special cases of the event's implementation proposals are as follows. In [S13], the author provides freedom to implement them, [S7] only considers the event `KeyEvent` of Java and in [S8] a different class is defined for each type of event (*change* events, *call* events, *time* events, and *completion* events) and for each *signal* event.

The authors more or less agree on the translation of *guards*, checking them directly as *if-statements* normally in the event methods or event-handler functions ([S2], [S6], [S7], [S13], [S18], [S25], and [S28]). Additionally, in [S8], [S9], [S15], and [S27] each guard is checked in an evaluated method, and in [S14] a specific class is created for guards.

Transition actions, on the other hand, are implemented as:

- A1: a method in the context class used by the vast majority of the proposals ([S2], [S6], [S8], [S9], [S10], [S11], [S13], [S15], [S18], [S19], [S20], [S25], and [S27]).
- A2: an interface which has to be implemented by all actions in the state machine specification ([S12]).
- A3: an action class with a method for each action ([S28]).
- A4: a single class ([S14]).

As noted above, the elements of the second and third group of Tables 10 and 11 are considered by few to almost none of the authors. Particularly noteworthy is the second group corresponding to *composite states* (hierarchy and concurrency), which are two of the main components of the expressive richness of state machines [5,78] and also improve their comprehension [79]. In particular, regarding *simple composite states* (hierarchical states), only [S2], [S3], [S7], [S10], [S13], [S15], [S18], [S19], [S20], [S21], [S25] and [S27] provide a proposal for their implementation, whereas others ([S8] and [S14]) state that they translate simple composite states but do not explain the strategy they use. A similar issue occurs with *orthogonal composite states*, which are implemented only by [S2], [S3], [S10], [S15], [S19], [S25], and [S27].

Finally, the third group corresponds to *pseudostates* and *activities*, which can be considered as secondary elements that, in our opinion, contribute to the complexity of models [78] without being essential. Regarding *pseudostates*, only [S8] takes into account all of them, defining a specific class for each type of pseudostate. On the other hand, [S25] takes into account all but one of them, using different strategies for their implementation (`fork` method, chained `if-statements` or references).

Finally, as far as *state activities* (we refer to entry/exit/do actions), they are implemented as:

- A1: a specific method in the context class ([S3], [S10], [S13], [S15] and [S20] consider *entry/exit actions* and [S9], [S18] and [S27] consider *entry/exit/do actions*).
- A2: a specific method in the event handler ([S21] implements *entry/exit actions*).
- A3: a specific method in the corresponding state class ([S3], [S19], [S25] and [S28] considers *entry/exit actions*).
- A4: a class for all actions ([S12] implements *entry/exit/do actions*).

3.4. Feature-based comparison

As described above, taking into account the RQ3 research question, first we determined a taxonomy of software development features expected to be considered by the published proposals. Then, we carried out a feature-based comparison that allows us to analyse and compare the selected proposals from a different point of view to that of the *pattern-based* and the *element-based* comparisons.

In Subsection 2.6 of Section 2, we present the taxonomy of 13 data items extracted from the selected papers classified into five groups. This taxonomy has been established taking into account: (i) the main software development features highlighted in the analysed proposals, and (ii) the list of system quality attributes presented in [80], which are based on their relevance to design patterns. In particular, we have taken [80] as a starting point and chosen the quality attributes we considered most appropriate for our particular study. We have also considered some comparison aspects directly related to state machines expressivity. Below, we explain in detail the features we have chosen for our taxonomy as well as the criteria followed to evaluate these features for each proposal.

- *Features related to state machines expressivity*
  - *Hierarchy*: whether hierarchical composite states are implemented or not, evaluated following the criteria: *Not Supported*, *Partially Supported*, or *Supported*.
  - *Concurrency*: whether orthogonal states are implemented or not, assessed by means of the criteria: *Not Supported*, *Partially Supported*, or *Supported*.

  – *History*: whether history pseudostates are implemented or not, using the criteria: *Not Supported*, *Partially Supported*, or *Supported*. We have included this feature in the taxonomy due to the fact that the history mechanisms, provided by statecharts [5] and its variants, are not provided by other state-based formalisms. This feature allows us to evaluate whether the authors make use of it or not.
- *Features related to software design*
  – *Expandability*: degree to which a system design can be extended [80], evaluated as *Bad*, *Medium*, or *Good*. From our point of view, it is closely related to the maintenance, extensibility, or flexibility aspects of the generated code.
  – *Simplicity*: degree to which the design of a system can be easily understood [80], assessed by means of the criteria: *No*, *Neutral*, or *Yes*.
  – *Reusability*: whether some code extracts resulting from the implementation of a specific state machine specification could be reused as part of the code generated from another, somehow related, state machine specification [80], assessed as: *Bad*, *Medium*, or *Good*.
- *Features related to implementation*
  – *Learnability*: degree to which the code source of a system is easy to learn [80], assessed by the criteria: *Bad*, *Medium*, or *Good*.
  – *Understandability*: degree to which the code source can be understood easily [80], evaluated as *Bad*, *Medium*, or *Good*. We think that the understandability, as well as the readability, of a software system is crucial for its reuse and evolution, so we have considered this feature as essential for evaluating the implementation proposals.
  – *Modularity*: degree to which the implementation of a system's functions are independent of one another [80], evaluated using *Bad*, *Medium*, or *Good*. It is worth highlighting that modularity is closely related to compactness and a more structured code. In addition, a modular implementation can result in, among other things, a simplified design, improved understandability and quality, and expedited response to system requirement changes.
- *Features related to runtime*
  – *Performance of execution*: how well the generated system is executed, evaluated as *Bad*, *Medium*, or *Good*. It is related to the speed of execution or the time the application takes to perform a requested task during execution.
  – *Memory needs*: amount of computer memory needed for the system's execution, assessed as *Low*, *Medium*, or *High*.
  – *Efficiency*: how well the system utilizes processor capacity, disc space, and memory, assessed by the criteria: *Bad*, *Medium*, or *Good*. Computational efficiency is related to both memory requirements and execution speed.
- *Feature related to the final result*
  – *Tooling*: whether the authors provide a tool implementing their proposal. In this case, we present the tool in question.

Finally, the results of this *feature-based* comparison are presented in Table 12, for pattern-based proposals, and in Table 13, for proposals not based on any pattern. We should mention that to evaluate a proposal regarding a specific feature, we have used two main aspects as a basis: (i) the papers themselves presenting such a proposal and on the way they present it, and (ii) other works that compare their approach with this proposal. In each cell of these tables, we have included the evaluation according to the scale used for the corresponding feature together with the source from which we have concluded the assessment. We should note that, in some cases, we have not found any clue in the paper's text that would give us a clear feature evaluation. In those situations, we have provided our own evaluation (represented in the table in grey cells); otherwise, we have represented it with an empty cell.

We should point out that there were several features that we needed to evaluate ourselves due to poor or non-existent information in the studies. Moreover, we would like to highlight that, for the *Expandability* and *Understandability* features of the proposal given by Gamma et al. [S11], studies do not agree on the evaluation criteria. Although we have decided to present the two opinions in Table 12, we have to say that we consider that the proposal given by Gamma et al. [S11] provides good understandability and maintenance. Similarly, studies do not agree on the assessment of the proposals given by Samek [S20] and by Yacoub et al. [S28] regarding *Understandability*, where in both cases we lean towards the assessments made by [S1] (particularly in [36]). Taking advantage of this remark, we would also like to mention that our feature-based results for the *State pattern* approach given by Gamma et al. [S11] (taking into account the second evaluations of *Expandability* and *Understandability* features) fits with the evaluation of this pattern given in [81], where the authors evaluate the impact of Gamma et al.'s 23 patterns.

Similarly, regarding tool development, we note that in this feature we have also considered the case of the tool Rhapsody [27] which, although not developed by Douglas [S9], uses switch statements as part of its code generation proposal.

4. Discussion

In this section we address three main issues. First, we summarize the principal findings of the systematic review, highlight the strengths and weaknesses of the evidence gathered, and discuss the relevance and contribution of the different techniques and implementation methods of state machine specifications into code published in the literature. Second, we discuss the study limitations. Finally, we analyse the threats to validity arising from the procedures we followed to perform the systematic review.

4.1. Principal findings

The main goal of this paper was to identify and classify the research work done in the field of code generation from state machine specifications and provide an exhaustive analysis and comparison of the proposals. To do so, a systematic review was conducted through an accurate process carrying out a three-based dimension comparison that in turn provides an answer to each of the given research questions. Next, we address each research question starting by discussing the main techniques and implementation methods proposed within the research topic (`RQ1`). The second subsection is devoted to highlighting the main state machine elements supported by the selected studies (`RQ2`), and the third subsection discusses the main desired software development features considered by the selected studies (`RQ3`).

4.1.1. Main techniques or implementation methods published within the field of state machines code generation

The review has led us to conclude that there is a strong tendency to use design patterns for implementing state machine specifications. Especially remarkable is the use of the *State Design pattern* (study [S11]), which is very appreciated in the field since it can be used in practically any size application [32]. Nevertheless, its lack of support for implementing certain aspects (such as the dynamic parts of the model or several state machines elements) has prompted the development of a great variety of extensions and patterns built upon this pattern in order to provide solutions for some of its weaknesses. This is the case of study [S25] by Tanaka et al., which provides one of the most complete approaches,

Table 12
Feature-based comparison of pattern-based proposals.

Criteria	Strategies/ References	State pattern		FSM Framework	State-Table pattern			Basic Statechart pattern	State Machine Design pattern	Statechart design pattern	Hierarchical State Machine pattern		QHsm pattern		EQHsm pattern	Reflective State Pattern
		Gamma et al. S11	(extension) Tanaka et al. S25	Gurp et al. S12	Douglass S9	(extension) Duby S10	(extension) Köhler et al. S15	Yacoub et al. S28	Shalyto et al. S23	Tomura et al. S27	Samek et al. S21	(extension) Heinzmann S13	Samek S20	(extension) Babitsky S3	Pintér et al. S19	Lamour et al. S17
State machine specifications' expressivity features	Hierarchy	**Not supported** [4,8,12,57,58]	**Supported** [6,7,8,68,69,70]	**Not supported** [4,8]	**Not supported** [12,46,47,57,58]	**Supported** [48,49]	**Supported** [13,14]	**Not Supported** [6,7,8,72]		**Supported** [8,12]	**Supported** [8]	**Supported** [50]	**Supported** [8,18,57,58]	**Supported** [39]	**Supported** [8,57,58,59,60]	**Not Supported** [54,55]
	Concurrency	**Not supported** [4,8,12,57,58]	**Supported** [6,7,8,68,69,70]	**Not supported** [4,8]	**Not supported** [12,57,58]	**Supported** [48,49]	**Supported** [13,14]	**Not Supported** [6,7,8,72]		**Supported** [8]	**Not supported** [8]	**Not supported** [50]	**Not supported** [8,57,58]	**Supported** [39]	**Supported** [8,57,58,59,60]	**Not Supported** [54,55]
	History	**Not supported** [48,58]	**Partially Supported** (shallow history) [6,7,8,68,69,70]		**Not supported** [46,47,57,58]			**Not Supported** [8,72]		**Not supported** [8]	**Partially supported** (shallow history) [63]	**Not supported** [50]	**Partially supported** (deep history) [57,58]		**Supported** [8,57,58,59,60]	
Software design features	Expandability	**Good** [56] **Bad** [4]	**Good** [6,7,8,68,69,70]		**Bad** [36,63,62]	**Medium**	**Medium**	**Good** [36,72]	**Good**	**Medium**	**Good** [63]	**Medium**	**Medium** [36,62]	**Medium**	**Medium** [57,58,59,60]	**Medium** [54,55]
	Simplicity	**Yes**	**Yes**	**Yes** [36]		**Neutral**	**Neutral**	**Yes**	**Yes**	**Yes**	**Neutral**	**Neutral**	**Yes** [18,36]	**Neutral**	**Neutral**	**Neutral**
	Reusability	**Bad** [4]	**Medium**			**Bad**	**Bad**		**Good** [66]	**Medium**						**Good** [54,55]
Implementation features	Learn ability	**Good**	**Good**	**Good**		**Medium**	**Medium**		**Good**	**Medium**	**Medium**	**Medium**	**Good** [36,62]	**Medium**	**Medium**	**Good**
	Understandability	**Bad** [12] **Good** [4,70]	**Good** [6,7,8,68,69,70]	**Good**	**Bad** [7,36,63,62,57,58]	**Medium**	**Medium**	**Bad** [6,7] **Good** [36]	**Good**	**Good** [12]	**Medium**	**Medium**	**Good** [18,62] **Bad** [36]	**Medium**		**Good** [54,55]
	Modularity	**Good** [70]	**Good** [6,7,8,68,69,70]		**Good** [36]	**Good** [49]	**Medium**		**Good**	**Medium**	**Good** [63]	**Medium**	**Good** [18,62]		**Good** [57,58,59,60]	**Good** [54,55]
Runtime features	Performance of execution	**Bad** [12,57,58]		**Good** [4,8]	**Good** [12]	**Good** [49]				**Bad** [12]	**Good** [63]	**Good** [50]	**Good** [36]			**Medium** [36,54,55]
	Memory needs	**High** [56,57,58]	**High**	**Medium** [8]	**High** [8,36,63,56,57,58]	**Medium** [49] (gives a proposal that saves space)					**Medium** [63]	**Low** [50]	**Low** [36,57,58]		**Low** [57,58,59,60]	
	Efficiency	**Good** [63]	**Good**		**Good** [56]						**Good** [63]	**Good** [50]	**Good** [62]		**Good** [57,58,59,60]	**Bad** [36]
Final result feature	Tooling		**JCODE** [6,7,8,68,69,70]	**FSMGenerator** [4,8]			**FUJABA** [13,14]									

Table 13
Feature-based comparison of not pattern-based proposals.

	Strategies/ References — Criteria	Switch statements		Knapp et al.	Shlaer et al.	Culwin	Chow et al.	Derezinska et al.	Lafreniere	Ali
		Douglass [57]	Metz et al. S18	S14	S24	S7	S6	S8	S16	S2
State machine specifications' expressivity features	Hierarchy	Not supported [12,57,58]	Supported [56]	Supported [51,52]	Not Supported [8,67]	Supported [41]		Supported [42,43]		Supported [11]
	Concurrency	Not supported [12,57,58]	Not supported [8]	Supported [51,52]	Not Supported [8,67]			Supported [42,43]		Supported [11]
	History	Not supported [57,58]	Partially supported (shallow history) [56]					Supported [42,43]		
Software design features	Expandability	Bad [6,8,36,57,58]	Bad [7,4,57,58, 63]			Bad	Bad	Bad	Medium	Medium
	Simplicity	Yes [50]	Yes [56]			Bad	Medium	Medium	Medium	Medium
	Reusability	Bad [6,8,57,58]	Bad [4]			Bad	Bad	Bad	Bad	Bad
Implementation features	Learn ability	Good				Good	Good	Good	Good	Good
	Understandability	Bad [6,8,36]	Bad [7,12]			Good	Good	Medium	Medium	Good
	Modularity	Bad [6,8]	Bad [56]	Medium		Bad	Bad	Bad	Bad	Bad
Runtime features	Performance of execution		Bad [12]							Good [11]
	Memory needs		Low [57,58]			Low	Low	Medium	Low	Low
	Efficiency									Good [11]
Final result feature	Tooling	Rhapsody [57,58]		HUGO [51,52,75]				FXU Framework [42,43]		

solving the *State pattern*'s substate problems, among other improvements. The *Reflective State pattern* (study [S17]) solves some of the design decisions chosen in the *State pattern* or even some disadvantages by allowing state subclasses themselves to specify their successor states. Patterns such as the *Basic Statechart pattern* (study [S28]) improve the *State pattern* by adding support for `guards` and `entry/exit actions`.

Other patterns, such as the *Finite State Machines pattern* (study [S12]), address the *State pattern*'s problems related to the evolution of state machines and data management by providing more structure at the implementation level. Another extension of the *State pattern* that focuses on reusability is the *State Machine Design pattern* (study [S23]), which makes the classes more reusable than the ones defined with the *State pattern*.

In the *Hierarchical State Machine pattern* (HSM) (study [S21]), state machines are defined, unlike the *State pattern*, by composition rather than by inheritance. The *Quantum Hierarchical State Machine pattern* (QHsm) (study [S20]) improve the HSM pattern by taking advantage of the Quantum Programming paradigm, enabling state machine specifications to be implemented in C or C++ by combining the HSM pattern and active-object based framework.

The *State-table pattern* (study [S9]), on the other hand, gives an approach to implement state machine specifications based upon a state table with references to concrete states and transitions.

Regarding proposals not pattern-based, it is worth citing the use of *nested switch statements*, which is one of the most common proposals to implement state machine specifications that works well with flat state machines but does not provide good solutions when hierarchy is presented in the state machine. This proposal is quite popular because it provides fast execution but at the cost of weaker readability and maintainability [6,8,36,57,58]. Several studies have used switch statements as the basis for state machine implementations. Among them it is worth citing Metz et al.'s work (study [S18]), which uses an ordinary switch statement to perform the system's state transition process. *Lafreniere*'s proposal (study [S16]), on the other hand, provides a solution to the problems arising in the use of switch statements by including support for both internal and external events, event data, and state transition validation. *Ali*'s proposal (study [S2]) provides a way of implementing concurrent and hierarchical state machines into efficient and encapsulated code based mainly on the use of Java `Enums`.

4.1.2. State machine elements supported by the selected studies

From the analysed studies, we have concluded that the subset of elements considered by almost every proposal is precisely the most widely used state machine elements (the *context class*, the *current state*, the *state-transition process*, *guards*, *simple states*, *events*, and *actions*). As for the implementation of the first four elements, authors more or less agree on the strategy followed since their semantics can be easily mapped in the usual language supporting concepts. This is not the case of, for example, *simple states* that are implemented in a variety of manners, not providing a clear trend for a unique implementation.

In some cases, the authors clearly opt for one or two specific implementation proposals. This is the case of *events* where, from the 18 studies that provide support for the implementation of events, 9 studies follow the approach labelled `E1` and 9 studies consider implementation `E2`. On the other hand, in the case of *Transition actions*, there are 17 studies that provide four different proposals (`A1–A4`), among which most of the works (13 studies) follow method `A1`.

Other elements are supported by very few studies. This is the case of, for example, *fork* and *join* (supported by two studies providing two different proposals), *choices* (supported by three studies providing two different implementation proposals), *shallow history* (supported by six studies providing three different proposals) and *deep history* (supported by four studies with two different proposals).

As for *composite states*, it is especially remarkable that not only are there few studies that provide an implementation, but also that these approaches are quite different from one another, going from more structured ones, based on classes and hierarchies (such as these by studies [S3], [S13], [S15], [S21], [S25], and [S27]) to simpler options, based on member functions or enumeration variables (such as these by studies [S2], [S7], [S10], [S18], [S19], and [S20]).

Another general conclusion that can be extracted from this comparison is that the proposal of Tanaka et al. [S25] is one of the most complete approaches. Furthermore, another key finding of this *element-based* comparison is that there are very few papers that support two of the main components of the expressive richness of state machine specifications [5,78] – hierarchy and concurrency.

4.1.3. Desired software development features considered by the selected studies

Our third research question addressed the desired features in software development provided by the selected studies. The first conclusion that can be extracted from our analysis (summarized in Tables 12 and 13) is that there are many papers that do not provide an implementation strategy taking into account relevant qualitative aspects in software development such as *maintenance* ([S6], [S7], [S8], [S9], [S18]), *reusability* ([S2], [S6], [S7], [S8], [S9], [S10], [S11], [S15], [S16], [S18]), and *modularity* ([S2], [S6], [S7], [S8], [S9], [S16], [S18]).

Another key finding concluded from this comparison is that the proposal of Tanaka et al. [S25] is one of the most complete, followed by Samek et al. [S21], Samek [S20] and Pintér et al. [S19], which are characterized by, on one hand, being easy to learn and to understand and, on the other hand, by the use of modular, structured code, decreased process memory requirements, and enhanced execution speed.

Regarding tool development, there are few works providing a tool that implements their proposal. Among them, *JCode* [S25], *FSMGenerator* [S12], *Rhapsody* [S9], and *FXU Framework* [S8] have been designed for state machine specifications to code generation, whereas the *Hugo* tool [S14] and the *Fujaba* tool [S15] were developed for broader purposes. Specifically, [S14] presents *Hugo*, which is described as a UML model translator for model checking, theorem proving, and code generation, but whose main efforts are oriented towards verification rather than code generation, so it is not aimed at producing product quality and optimized code.

As concerns study [S25], the authors have successfully implemented their approach in their system, *JCode*, which aims to automatically convert UML class and statechart diagrams specifications into Java code. This tool has mainly been evaluated by comparing it with the Rhapsody tools, showing that the code generated by the *JCode* system is about 68% more efficient and about four times more compact than that of *Rhapsody*.

As for the *FSMGenerator* tool, presented in [S12], the authors evaluate their approach by comparing it with the *State pattern* with respect to maintenance and performance, obtaining promising results in maintenance and quite an acceptable performance.

The *FXU Framework* is presented in study [S8] as the first solution to support the generation and execution of all elements of state machine UML 2.0 using C# language. The authors have assessed the proposal by performing several experiments showing that an application performing a behaviour specified in state machine models can be developed effectively and reliably [43]. In particular, the code generation was tested on over forty models (classes and state machines), using all possible constructs of UML 2.x state machines in different situations, including elements such as complex and orthogonal states, different kinds of pseudostates, and submachine states. Moreover, the programs performing those state machines were run taking into account different sequences of triggering events.

Last, the *Fujaba* tool (study [S15]), originally aimed at supporting software forward and reverse engineering (it is an acronym for ''From UML to Java and back again''), has nowadays become a complex project with support for model-based software engineering and re-engineering.

4.2. Study limitations

This study has the usual limitations associated with any systematic literature review. With respect to the search process, we have to admit that we may have missed some relevant papers and the data extraction may have been incorrect for some papers. In this process, we have limited ourselves to English-language studies and to five major electronic databases using search terms related to specific English terms. Therefore, this strategy did not take into account non-English-language papers, papers in many national journals and conferences, or papers that use keywords not considered or unusual terminology. Overall, though, we do not expect to have missed a large number of important studies.

As regards data extraction, we should also mention that some aspects of the literature, related to both the *element-based* and the *feature-based* comparisons, have been rooted on our own interpretation of the papers (especially in the latter comparison), so it is possible that other researchers might arrive at different conclusions.

4.3. Threads to validity

A systematic literature review such as this one has several evident threats to its validity. First, whether we have identified adequate keywords and chosen enough search engines. In this respect, the ample list of different papers indicates that the width of the search is sufficient. The snowball sampling procedure shown that the search worked well since we have not found any additional papers conforming to the established inclusion criteria.

Second, another possible threat to validity is bias in applying quality assessment and data extraction. In order to minimize this threat insofar as possible, we explicitly established the inclusion and exclusion criteria, which we believe was detailed enough to provide an assessment of how we selected the chosen papers for analysis.

Finally, another important threat to validity is reliability, which focuses on whether the data are extracted and the analysis is performed in such a way that it can be repeated by other researchers to obtain the same results. In this respect, we have defined search terms and the procedures applied during the review so that it may be replicated by others, with the exception of, as described in Subsection 4.2, those aspects considered in the *element-based* and the *feature-based* comparisons, where we have based on our own interpretation of the papers.

5. Conclusions

In this paper we provide a systematic literature review of peer-reviewed published studies that focus on the code generation from state machine specifications. For this task, the review presented herein has been conducted using an accurate and reliable process that has allowed us to identify, analyse, and compare a comprehensive set of 53 relevant proposals published within the research topic.

The systematic review has addressed the following main research questions: (RQ1) What techniques or implementation methods have been used for generating object oriented code from state machine specifications (including UML state machines, finite state machines and Harel statecharts)?, (RQ2) What state machine specification elements are translated into code by each technique or implementation method? and (RQ3) What are the desired software development features considered by the published proposals? the main findings of the review being those presented in the following paragraphs.

Regarding the first question, the results of the review show that the different techniques for code generation from state machine specifications can be mainly classified as those based on a design pattern and those that are not. More specifically, the selected papers provide a total of 23 different implementation proposals (15 corresponding to pattern-based approaches and 8 related to not pattern-based approaches), which are used in a wide variety of application contexts such as in distributed control systems or even in NASA space missions. In particular, we would like to make a special comment regarding the *State design pattern*, which constitutes one of the most widely used patterns to model the execution semantics and which has been taken as the basis for the development of a wide number of pattern-based proposals.

As regards the second question, we found that most of the analysed works provide an implementation proposal of the main state machine elements commonly used for the representation of the dynamic behaviour of systems (such as simple states, events, guards, and actions in transitions), whereas only a few proposals support implementation of more specific elements such as simple and orthogonal composite states, which are two of the main components of the expressive richness of state machine specifications [5,78]. In addition, another conclusion drawn from the review is that the proposal of Tanaka et al. [S25] is one of the most complete approaches.

Regarding the third research question, a key finding of the review obtained from the *feature-based* comparison is that there

Table A.14
Search strings.

DB	Place of search	Detailed search strings	Num.
IEEE Xplore	Abstract (summary) and title text, and indexing terms	(''UML Statecharts'' OR ''UML State Machines'' OR ''Harel's Statecharts'' OR ''State charts'' OR ''Finite State Machines'' OR ''FSM'' OR ''Dynamic models'' OR ''State-based behavior'' OR ''State-based behaviour'' OR ''Dynamic behavior'' OR ''Dynamic behaviour'' OR ''Reactive system'') **AND** (''Implementation'' OR ''Code generation'' OR ''Model-based generation'' OR ''Mapping'' OR ''Executable specification'' OR ''Behavioral patterns'' OR ''Behavioural patterns'' OR ''Design patterns'' OR ''Object-oriented language'' OR ''Programming language'' OR ''Object-oriented method'' OR ''OO method'' OR ''State-Oriented programming'' OR C++ OR Java OR C/C++)	*1384*
ACM	Title, abstract and keywords	Being: **A** = ''UML Statecharts'' OR ''UML State Machines'' OR ''Harel's Statecharts'' OR ''State charts'' OR ''Finite State Machines'' OR ''FSM'' OR ''Dynamic models'' OR ''State-based behavior'' OR ''State-based behaviour'' OR ''Dynamic behavior'' OR ''Dynamic behaviour'' OR ''Reactive system'' and **B** = ''Implementation'' OR ''Code generation'' OR ''Model-based generation'' OR ''Mapping'' OR ''Executable specification'' OR ''Behavioral patterns'' OR ''Behavioural patterns'' OR ''Design patterns'' OR ''Object-oriented language'' OR ''Programming language'' OR ''Object-oriented method'' OR ''OO method'' OR ''State-Oriented programming'' OR C++ OR C# OR Java OR C/C++ The search strings were:	
		(Title: (A) AND Title: (B)) OR (Abstract: (A) AND Abstract: (B))	690
		(Keywords: (A) AND Keywords: (B))	31
		Total	*721*

ScienceDirect	Abstract, title, keywords	Being:
		A = {UML Statecharts} OR {UML State Machines} OR {Harel's Statecharts} OR {State charts} OR {Finite State Machines} OR FSM OR {Dynamic models}
		B = {State-based behaviour} OR {State-based behaviour} OR {Dynamic behaviour} OR {Dynamic behaviour} OR {Reactive system}
		C = Implementation OR {Code generation} OR {Model-based generation} OR Mapping OR {Executable specification} OR {Behavioral patterns} OR {Behavioural patterns} OR {Design patterns}
		D = {Object-oriented language} OR {Programming language} OR {Object-oriented method} OR {OO method} OR {State-Oriented programming} OR C++ OR C# OR Java OR C/C++
		The search strings were:	

		TITLE-ABSTR-KEY (A) AND TITLE-ABSTR-KEY (C)	157
		TITLE-ABSTR-KEY (A) AND TITLE-ABSTR-KEY (D)	19
		TITLE-ABSTR-KEY (B) AND TITLE-ABSTR-KEY (C)	75
		TITLE-ABSTR-KEY (B) AND TITLE-ABSTR-KEY (D)	9
		Total	*260*

Table A.15
Search strings.

DB	Place of search	Detailed search strings	Num.
Microsoft Academic Search	Anywhere in the article	Considering all possible pair permutations of the first group of keywords chosen in the first step of the electronic systematic search, that is, UML Statecharts, Harel's Statecharts and Reactive Systems, and Implementation, Code Generation, and Programming language	*1184*
Google Scholar	Title	Being: **A** = ``UML Statecharts'' OR ``UML State Machines'' OR ``Harel's Statecharts'' OR ``State charts'' OR ``Finite State Machines'' OR FSM OR ``Dynamic models'' **B** = ``State-based behavior'' OR ``State-based behaviour'' OR ``Dynamic behavior'' OR ``Dynamic behaviour'' OR ``Reactive system'' **C** = ``Implementation'' OR ``Code generation'' OR ``Model-based generation'' OR ``Mapping'' **D** = ``Executable specification'' OR ``Behavioral patterns'' OR ``Behavioural patterns'' OR ``Design patterns'' **E** = ``Object-oriented language'' OR ``Programming language'' OR ``Object-oriented method'' OR ``OO method'' **F** = ``OO-method'' OR ``State-Oriented programming'' OR C++ OR C# OR Java OR C/C++ The search strings were:	
		allintitle: (A AND C)	50
		allintitle: (A AND D)	1
		allintitle: (A AND E)	0
		allintitle: (A AND F)	5
		allintitle: (B AND C)	8
		allintitle: (B AND D)	0
		allintitle: (B AND E)	0
		allintitle: (B AND F)	1
		Total	*65*

are many papers that do not provide an implementation strategy that takes into account relevant qualitative aspects in software development such as maintenance, reusability, or modularity, whereas others, such as those of Tanaka et el. [S25], Samek et al. [S21], Samek [S20] and Pintér et al. [S19], particularly care about desirable features to be considered in software development.

As another general conclusion drawn from the review, we have to say that UML state machines are the most common form of state machine specification used in code generation studies.

The classification of the literature should be of value both to researchers and software developers who need a satisfactory solution for the implementation of the behavioural models representing the dynamics of their system. In particular, the work carried out in this review has been taken as a starting point for the development of our own proposal for the implementation of UML Statecharts in a specific medical context [35,82,83], based on the proposal by Tanaka et al. [S25].

Acknowledgments

This work has been partially supported by the Spanish Ministry of Science and Innovation (projects TIN2009-13584 and SMOTY (IPT-2011-1328-390000)) and the Centre for Industrial Technological Development (CDTI) (project THOFU (CEN20101019)).

Appendix A. The search strings

The final abstract search string constructed containing all the relevant keywords chosen for the search is: (``UML Statecharts'' OR ``Harel's Statecharts'' OR ``Reactive system'' OR ``UML State Machines'' OR ``State charts'' OR ``Finite State Machines'' OR ``FSM'' OR ``State-based behavior'' OR ``State-based behaviour'' OR ``Dynamic models'' **OR** ``Dynamic behavior'' OR ``Dynamic behaviour'') AND (``Implementation'' OR ``Code generation'' OR ``Programming language'' OR ``Object-oriented language'' OR ``OO language'' OR C++ OR C# OR Java OR C/C++ OR ``State-Oriented programming'' OR ``Object-oriented method'' OR ``OO method'' OR ``Model-based generation'' OR ``Mapping'' OR ``Executable specification'' OR ``Behavioral patterns'' OR ``Behavioural patterns'' OR ``Design patterns'').

As we described above, starting from such an abstract search string, different strings were derived on each database engine, taking into account their search particularities. These strings are explained in Tables A.14 and A.15, where we have specified: (1) the electronic database in which the search took place, (2) the fields in which the search took place, (3) the detailed search string(s), and (4) the number of papers resulting from the search (hence, note that this number refers to papers without considering any selection criteria). Explaining in more detail, taking into account our particularly long search string, we had to use the advanced search option in all databases. More specifically, in some databases we came across several problems related to the length of our search string. The fact is that, as expected, these database engines have size limitations in their search text boxes, even in the advanced search. This issue led us to perform separate searches on these electronic databases (particularly in ACM, ScienceDirect, Microsoft Academic Search, and Google Scholar), considering different "equivalent" combinations of boolean search strings, hence obtaining duplicated results. That is the case of, for example, ScienceDirect (see Table A.14). In this database, we had to split the search string into four different strings (A and B with the OR concatenation of the State machine specification keywords, and C and D with the OR concatenation of the Software engineering concepts) and carried out the search of the four possible combinations using the AND to join these major terms. On the other hand, the search constraints in the case of Microsoft Academic Search were even more limiting, forceding us first to carry out the searches with the 12 × 18 = 216 possible pairs (one pair per each combination of the State machine specifications and the Software engineering concepts). This search obtained as a result a total of 35325, which we considered to be an unmanageable number. Therefore, we decided to perform the searches in this database considering all possible pair permutations of the first group of keywords chosen in the first step of the electronic systematic search, that is, UML Statecharts, Harel's Statecharts and Reactive Systems, and Implementation, Code Generation, and Programming language, yielding a total of 1184 results.

Regarding the search location in the papers (both in electronic databases and the specified conference proceedings), titles, abstracts, and keywords of the articles were searched (see Tables A.14 and A.15), with the exception of Google Scholar and Microsoft Academic Search. In particular, Google Scholar allows for searching either in the title or anywhere in the article, so, in order to be as similar as possible to the other databases, we searched in the paper titles. On the other hand, Microsoft Academic Search carries out the search in all the paper contents, being able to establish constraints only regarding the author, conference, journal, organization, year, and DOI. For this reason, we searched in the entire paper's contents.

Taking into account that all databases cover a wide range of years, we performed the searches choosing the default option *All available years*. Hence, since the search took place in October 2010, all resources up to the first half of 2010 were considered in the search. Finally, the research found in these electronic databases yielded a total of 3623 results.

References

[1] B. Selic, The pragmatics of model-driven development, IEEE Software 20 (5) (2003) 19–25.
[2] OMG Model Driven Architecture, Committed Companies & Their Products, 2010. <http://www.omg.org/mda/committed-products.htm> (last visited May 2012).
[3] OMG, UML 2.3 Superstructure Specification, Document Formal/2010-05-05, 2010. <http://www.omg.org/> (last visited May 2012).
[4] J.V. Gurp, J. Bosch, On the implementation of finite state machines, in: Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, 1999.
[5] D. Harel, Statecharts: a visual formulation for complex systems, Science of Computer Programming 8 (3) (1987) 231–274.
[6] I.A. Niaz, J. Tanaka, Code generation from UML statecharts, in: Proc. 7th IASTED Conf. on Software Engineering and Application (SEA 2003), 2003, pp. 315–321.
[7] I.A. Niaz, J. Tanaka, Mapping UML statecharts to java code, in: Proceedings of the IASTED Conf. on Software Engineering, 2004, pp. 111–116.
[8] I. Azim, Automatic Code Generation From UML Class and Statechart Diagrams, PhD Dissertation, University of Tsukuba, 2005.
[9] A. Jakimi, M. Elkoutbi, An object-oriented approach to UML scenarios engineering and code generation, International Journal of Computer Theory and Engineering (IJCTE) 1 (1) (2009) 35–41.
[10] A. Jakimi, M. Elkoutbi, Automatic code generation from UML statechart, International Journal of Computer Theory and Engineering (IJCTE) 1 (2) (2009) 165–168.
[11] J. Ali, Using Java Enums to implement concurrent–hierarchical state machines, Journal of Software Engineering 4 (3) (2010) 215–230.
[12] T. Tomura, S. Kanai, K. Uehiro, S. Yamamoto, Object-oriented design pattern approach for modeling and simulating open distributed control system, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2001), 2001, pp. 211–216.
[13] H.J. Köhler, U.A. Nickel, J. Niere, A. Zündorf, Using UML as a Visual Programming Language, Tech. Rep. tr-ri-99-205, University of Paderborn, Paderborn, Germany (August 1999).
[14] H.J. Köhler, U. Nickel, J. Niere, A. Zündorf, Integrating UML diagrams for production control systems, in: Proc. of the 22nd International Conf. on Software Engineering (ICSE 2000), 2000, pp. 241–251.
[15] R. Tiella, A. Villafiorita, S. Tomasi, Speciffication of the control logic of an eVoting system in UML: the ProVotE experience, in: Proceedings of the 5th International Workshop on Critical Systems Development Using Modeling Languages (CSDUML06), 2006, pp. 93–102.
[16] R. Tiella, A. Villafiorita, S. Tomasi, FSMC+, a tool for the generation of Java code from statecharts, in: Principles and Practice of Programming in Java, 2007, pp. 93–102.
[17] A. Villafiorita, K. Weldemariam, R. Tiella, Development, formal verification, and evaluation of an E-voting system with VVPAT, IEEE Transactions on Information Forensics and Security 4 (4) (2009) 651–661.
[18] E. Benowitz, K. Clark, G. Watney, Auto-coding UML statecharts for flight software, in: Proceedings of the Second IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06), 2006, pp. 413–417.
[19] K.P. Kiri L. Wagstaff, L. Scharenbroich, From Protocol Specification to Statechart to Implementation, JPL Technical Report, Nasa, Jet Propulsion Laboratory, 2008. http://ml.jpl.nasa.gov/papers/wagstaff/wagstaff-protocol-statecharts-08.pdf (last visited May 2012).
[20] K.L. Wagstaff, E. Benowitz, D.J. Byrne, K. Peters, G. Watney, Automatic code generation for instrument flight software, in: Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space, 2008.
[21] S. Teppola, P. Parviainen, J. Takalo, Challenges in deployment of model driven development, in: The Fourth International Conference on Software Engineering Advances (ICSEA 2009), 2009, pp. 15–20.
[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
[23] B. Kitchenham, Procedures for Performing Systematic Reviews, Technical Report tr/se-0401, Keele University, 2004. <http://www.eecis.udel.edu/~cisgsa/lib/exe/fetch.php?id=research&cache=cache&media=faq:kitchenham_2004.pdf> (last visited May 2012).
[24] B. Kitchenham, S. Charters, Guidelines for Performing Systematic Literature Reviews in Software Engineering, Version 2 (EBSE 2007-01) (2007) 200701. <http://pages.cpsc.ucalgary.ca/sillito/cpsc-601.23/readings/kitchenham-2007.pdf> (last visited May 2012).
[25] M. Petticrew, H. Roberts, Systematic Reviews in the Social Sciences: A Practical Guide, Blackwell Publishing, 2005.
[26] Gentleware, Poseidon for UML Case Tool, 2012. <http://www.gentleware.com/> (last visited May 2012).
[27] Rhapsody, 2012. <http://www.ibm.com/developerworks/rational/products/rhapsody/> (last visited May 2012).
[28] Borland Together 2008 Edition for Eclipse, 2012. <http://www.borland.com/us/products/together/> (last visited May 2012).
[29] Executable UML, UniMod, UniMod Project, 2008. <http://unimod.sourceforge.net/> (last visited May 2012).
[30] IBM, IBM Rational Rose, 2012. <http://www-01.ibm.com/software/awdtools/developer/rose/> (last visited May 2012).
[31] Tigris.org, Open Source Software Engineering Tools, ArgoUML Modeling Tool, 2012. <http://argouml.tigris.org> (last visited May 2012).
[32] T. Allegrini, Code Generation Starting from Statecharts Specified in UML, argoUML White Paper, 2002. <http://argouml.tigris.org/docs/allegrini_dissertation/tesi_en.pdf>. (last visited May 2012).
[33] I. Niaz, Automatic Code Generation From UML Class and Statechart Diagrams, PhD Dissertation, University of Tsukuba, 2005.
[34] K.O. Chow, W. Jia, V.P. Chan, J. Cao, Model-based generation of Java code, in: Proc. International Conf. On Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), 2000.
[35] B. Pérez, Towards Decision Facts Management Systems: The Particular Case of Clinical Guidelines, PhD Thesis, University of Zaragoza, Department of Computer Science and Systems Engineering, Zaragoza, Spain, 2011.
[36] P. Adamczyk, The anthology of the finite state machine design patterns, in: Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP), 2003.
[37] P. Adamczyk, Selected patterns for implementing finite state machines, in: Proceedings of the 11th Conference on Pattern Languages of Programs (PLoP), 2004.
[38] J. Ali, Implementing statecharts using Java enums, in: Proceedings of the 2nd International Conference on Education Technology and Computer (ICETC), vol. 4, 2010, pp. 413–417.
[39] D. Babitsky, Hierarchical state machine design in C++, C/C++ Users Journal. <http://drdobbs.com/cpp/184402040> (last visited May 2012).
[40] F. Chauvel, J. Jézéquel, Code generation from UML models with semantic variation points, in: Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Lecture Notes in Computer Science, vol. 3713, Springer, 2005, pp. 54–68.
[41] F. Culwin, The statechart design of a novel date input mechanism, Italics e-Journal 3 (1). <http://www.ics.heacademy.ac.uk/italics/Vol3-1/statechart paper/statechartpaper.PDF> (last visited May 2012).
[42] R. Pilitowski, A. Derezinska, Code generation and execution framework for UML 2.0 classes and state machines, in: Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, LNCS, Springer Verlag, 2007, pp. 421–427.
[43] A. Derezinska, R. Pilitowski, Correctness issues of UML class and state machine models in the C# code generation and execution framework, in: Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT'2008), 2008, pp. 517–524.
[44] A. Derezinska, R. Pilitowski, Event processing in code generation and execution framework of UML state machines, Software Engineering in Progress (2007) 80–92.
[45] A. Derezinska, R. Pilitowski, Realization of UML class and state machine models in the C# code generation and execution framework, Informatica (Slovenia) 33 (2009) 431–440.
[46] B.P. Douglass, Real Time UML – Developing Efficient Objects for Embedded Systems, Addison-Wesley, Massachusetts, 1998.
[47] B.P. Douglass, Class 505/525: state machines and statecharts, in: Proceedings of Embedded Systems Conference Fall, 2001.
[48] C. Duby, Class 265: implementing UML statechart diagrams, in: Proceedings of Embedded Systems Conference Fall, 2001.
[49] C. Duby, Implementing UML Statechart Diagrams, White Paper, 2004, <http://www.pathfindermda.com/wp-content/themes/pathfinder/downloads/implementing_state_charts.pdf> (last visited May 2012).
[50] S. Heinzmann, Yet another hierarchical state machine, Overload Journal: Association of C & C++ Users (64) (2004) 14–21.

[51] A. Knapp, S. Merz, Model checking and code generation for UML state machines and collaborations, in: G. Schellhorn, W. Reif (Eds.), Proceedings of the 5 th Workshop on Tools for System Design and Verification (FM-TOOLS), Institut für Informatik, Universität Augsburg, Reisensburg, Germany, 2002, pp. 59–64.

[52] A. Knapp, S. Merz, C. Rauh, Model checking – timed UML state machines and collaborations, in: Proceedings of the Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium (FTRTFT 2002), vol. 2469, Lecture Notes in Computer Science, 2002, pp. 395–416.

[53] D. Lafreniere, State machine design in C++, C/C++ Users Journal 18 (5) (2000) 58–66.

[54] L. Lamour, C.M.F. Rubira, The reflective state pattern, in: Proceedings of the Pattern Languages of Program Design, TR-WUCS-98-25, 1998.

[55] L. Lamour, C.M.F. Rubira, Reflective design patterns to implement fault tolerance, in: Proceedings of the Workshop on Reflective Programming in C++ and Java, OOPSLA'98, 1998, pp. 81–85.

[56] P. Metz, J. O'Brien, W. Webern, Code generation concepts for statechart diagrams of the UML v1.1, in: Proc. of the Object Technology Group (OTG) Conference, 1999.

[57] G. Pintér, I. Majzik, Program code generation based on UML statechart models, Periodica Polytechnica, Electrical Engineering 47 (3–4) (2003) 087–204.

[58] G. Pintér, I. Majzik, Automatic code generation based on formally analyzed UML statechart models, in: Proceedings of the Workshop on Formal Methods for Railway Operation and Control Systems, Budapest, L'Harmattan Kiad, 2003, pp. 45–52.

[59] G. Pintér, I. Majzik, Impact of statechart implementation techniques on the effectiveness of fault detection mechanisms, in: Proceedings of the 30th EUROMICRO Conference, IEEE Computer Society, 2004, pp. 136–143.

[60] G. Pintér, Model Based Program Synthesis and runtime Error Detection for Dependable Embedded Systems, PhD Thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, Budapest, Hungary, 2007.

[61] G. Pintér, I. Majzik, Code Generation Based on UML Statecharts, in: Proceedings of the 10th PhD Mini-Symposium: Budapest University of Technology and Economics Department of Measurement and Information Systems, 2003, pp. 18–19.

[62] M. Samek, Practical Statecharts in C/C++, Quantum Programming for Embedded Systems., CMP Books, 2002.

[63] M. Samek, P. Montgomery, State-oriented programming, International Journal of Embedded Systems 13 (8) (2000) 22–43.

[64] M. Samek, Practical Statecharts in C/C++, CMP Books, 2002.

[65] A.V. Saúde, R.A.S.S. Victorio, G.C.A. Coutinho, Persistent state pattern, in: Proceedings of the 17th Conference of Pattern Languages of Programs (PLoP), 2010.

[66] A. Shalyto, N. Shamgunov, G. Korneev, State machine design pattern, in: Proceedings of the 4th International Conference on.NET Technologies, 2006, pp. 51–57.

[67] S. Shlaer, S.J. Mellor, Object Lifecycles Modeling The World in States, Addison-Wesley, 1992.

[68] J. Ali, J. Tanaka, Converting statecharts into Java code, in: Proceedings of the 5th International Conference on Integrated Design and Process Technology (IDPT'99), 1999, pp. 111–116.

[69] J. Ali, J. Tanaka, Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams, Journal of Computer Science and Information Management (JCSIM) 2 (1) (2001) 24–34.

[70] I.A. Niaz, J. Tanaka, An object-oriented approach to generate Java code from UML statecharts, International Journal of Computer & Information Science 6 (2) (2005).

[71] T. Tomura, S. Kanai, K. Uehiro, S. Yamamoto, Developing simulation models of open distributed control system by using object-oriented structural and behavioral patterns, in: Proceedings of the fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Computer Society, 2001, pp. 428–437.

[72] S.M. Yacoub, H.H. Ammar, A pattern language of statecharts, in: Proceedings of the Fifth Annual Conf. on the Pattern Languages of Program (PLoP'98), 1998, pp. 98–99.

[73] D. Alur, D. Malks, J. Crupi, Core J2EE Patterns: Best Practices and Design Strategies, 2nd ed., Prentice Hall/Sun Microsystems Press, 2003.

[74] SPIN and PROMELA Reference Manual, 2011. <http://spinroot.com/spin/whatispin.html> (last visited May 2012).

[75] HUGO Tool, 2008. <www.pst.ifi.lmu.de/projekte/hugo> (last visited May 2012).

[76] OMG, UML 2.1.2 Superstructure Specification, Document Formal/2007-11-02, 2007. <http://www.omg.org/> (last visited May 2012).

[77] Sun Microsystems, The Java Tutorial, 2012. <http://download.oracle.com/javase/tutorial/java/javaOO/enum.html> (last visited May 2012).

[78] D.L. Moody, The ''physics'' of notations: toward a scientific basis for constructing visual notations in software engineering, IEEE Transactions on Software Engineering 35 (6) (2009) 756–779.

[79] D. Budgen, A.J. Burn, O.P. Brereton, B. Kitchenham, R. Pretorius, Empirical evidence about the UML: a systematic literature review, Software: Practice & Experience 41 (4) (2011) 363–392.

[80] F. Khomh, Y.G. Guéhéneuc, Do design patterns impact software quality positively? in: Proceedings of the 12th Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, 2008, 5pp (Short Paper).

[81] F. Khomh, Y.G. Guéhéneuc, DEQUALITE: building design-based software quality models, in: Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP), ACM Press, 2008.

[82] E. Domínguez, B. Pérez, M.A. Zapata, Towards a traceable clinical guidelines application: a model driven approach, Methods of Information in Medicine 46 (6) (2010) 571–580.

[83] B. Pérez, I. Porres, Authoring and verification of clinical guidelines: a model driven approach, Journal of Biomedical Informatics 43 (4) (2010) 520–536.