FSMC+, a Tool for the Generation of Java Code from Statecharts

Roberto Tiella
IRST
via Sommarive, 18
Trento, Italy
tiella@itc.it

Adolfo Villafiorita
IRST
via Sommarive, 18
Trento, Italy
adolfo@itc.it

Silvia Tomasi
IRST
via Sommarive, 18
Trento, Italy
sitomasi@itc.it

ABSTRACT

ProVotE is a two-phase project aiming at actuating art. 84 of law 2 - 5/3/2003 of the Autonomous Province of Trento (Italy), which promotes the introduction of e-voting systems for the next provincial elections in Trentino (Nov. 2008).

During the first phase of the ProVotE project we built *jprovote*, a Java/Linux e-voting system. The *jprovote* system has been used with experimental value by more than 11000 voters during local elections held in various municipalities of Trentino (Italy).

A critical component of *jprovote* is its core logic, that is responsible of controlling the overall behavior of the e-voting machine during an election. In order to simplify its development and to allow for formal verification of this critical component we developed FSMC+.

FSMC+ is a compiler that takes as input a subset of UML Statecharts and produces the corresponding Java and NuSMV code (NuSMV is a model checker developed at ITC-irst). Support for parameters in events, complex expressions in guards, and support to nested states are some of the distinguishing features of FSMC+.

In this paper we present FSMC+ and we show how we used it for the development and the verification of the ProVotE e-voting machine. Even though FSMC+ has been specifically created to ease the development of *jprovote*, we believe the approach and the tool we developed to be general enough to be used in other applications.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming

General Terms

Design, Reliability, Verification

PPPJ 2007, September 5–7, 2007, Lisboa, Portugal.

Keywords

evoting, statecharts, code generation, model checking

1. INTRODUCTION

The use of new technologies to support elections has been and is the subject of great debate. Not surprisingly, there are both advocates of the benefits it can bring — such as improved speed, impartiality in counting, and accessibility — and people more concerned with the risks it poses, such as digital divide, violations to secrecy and anonymity, alteration of the votes and of the results, either because of malicious attacks or simply because of bad design and coding. (See, for instance, [10, 23, 28, 6, 21] for a more in depth view on some of the topics mentioned above.)

The development of e-voting systems is particularly demanding. System development is subjected to strong deadlines (a delay of one day can make the difference between success and failure, as election days cannot be shifted), systems availability during election is a critical issue (in Italy most elections are run in one day, from 6.00am to 10pm, during which all voters must be given the right to express their votes when they show up at the polling station), and are under constant scrutiny by citizens, experts, and representatives of the parties.

The last few years have seen a wider adoption of tools and techniques to simplify the development and increase the quality of complex embedded systems. In various safety critical sectors, such as automotive, railway, avionics, the use of finite state machines for the specification of embedded controllers and the use of theorem provers and model checkers to increase confidence in the design have become more common. However, to our knowledge, the use of such techniques in the e-voting domain does not seem to be common practice.

The Autonomous Province of Trento (also PaT, from now on), which benefits of special autonomy and determines by its own legislation how the council and the President of the province are elected, is considering the introduction of e-voting for the next provincial elections as mandated by Art. 84 of PaT law 2/2003. To actuate the law PaT is sponsoring the ProVotE project[1], a two-phase project which has the goal of ensuring a smooth transition to the new way of voting, eliminating risks of digital divide, and providing the

[1]ProVotE is the acronym of "Progetto Voto Elettronico", e-Voting Project in Italian

technological solutions required for an electronic election. The project is organized in three lines of activities, sociological, normative, and technological, which strictly interact. (See [32, 9] for more details and [26] for some considerations related to the sociological aspects of e-voting.)

During the first phase of ProVotE we built *jprovote*, a Java/Linux e-voting system, which, so far, has been tested by more than 11000 citizens in various trials conducted in Trentino during local elections.

A critical component of *jprovote* is its core logic, whose main functions are:

- ensure that the e-voting machine correctly implements the procedures required by the Italian electoral law;
- coordinate and control all the devices of the voting machine (touchscreen, smartcard reader, printer);

Malfunctions and errors in *jprovote*'s core logic may compromise some fundamental principles of the law, such as allowing people to vote more than once, exposing the expression of a voter to other voters, or cause interruptions in the availability of e-voting machines during the election day.

In order to minimize risks related to the development of the core logic we decided to adopt some of the practices commonly used for the development of embedded systems. For the development of the core logic of *jprovote*, thus we adopted a model-driven approach [8] that supported code generation and formal verification using model checking [20, 13]. In parallel, we implemented a tool, called FSMC+, to support our development process.

FSMC+ is a compiler that takes as input a subset of the Statecharts [19] notation and produces the corresponding Java and NuSMV code. (NuSMV is a model checker developed at ITC-irst — see [11]). Similarly to other freely (and commercially) available tools, such as, e.g., Hugo/RT [30], SPIDER/Theseus environment [17], Statemate[2], FSMC+ implements a model-driven approach in which the the control logic of a system can be specified using a high-level notation and the code implementing such logic can be automatically generated and verified using model-checking. Support for UML Statecharts, parameters in events, complex expressions in guards, nested states, a template system for code generation and easier inspection of the performed steps are, however, some of the distinguishing features of FSMC+.

In this paper we present FSMC+ and we show how we used it for the development and the verification of the ProVotE e-voting system.

The rest of the paper is organized as follows. The next two sections provide some motivations for the implementation of FSMC+ and some information on how FSMC+ distinguishes from other similar tools. Section 4 presents FSMC+ and Section 5 a case study, namely the development of the *jprovote* system. We conclude with an analysis of the advantages and limits of the approach and with some considerations about future work.

[2] *http://modeling.telelogic.com/*

2. MOTIVATIONS

The development of FSMC+ within ProVotE has been driven by the following considerations:

- *applicative domain*: the (Italian) electoral norms provide detailed descriptions of the procedures that have to be followed and are particularly adapt at being represented as finite state machines. By designing the e-voting machine architecture to incorporate state machines in its core, we could provide a tighter link between the behavior of the machine and the Italian electoral law and, at the same time, allow for the use of formal verification techniques.
- *project constraints*: in order to meet the first deadline of the project (an experimentation during the the May 2005 election) the first prototype of the e-voting machine had to be developed under a rather tight schedule. In order to keep with the project deadlines we needed an approach and tools that could allow us to integrate state machines in our system "incrementally". The implementation of a tool which we could make grow together with and adapt to the project needs could allow us to address such issue. The development of FSMC+ thus, has proceeded in parallel to the development of *jprovote*, by first providing a set of core functions (i.e. translation form textual representation of state charts to Java code) and then adding more complex functions (such as a support for a graphical front-end) as the need arose.
- *verifiability and compliance to standards*: we wanted to be able to easily inspect the generated code and have the generated code conform to our coding guidelines.
- *"market" opportunity*: various characteristics of FSMC+ cannot be found in other freely available tools (see the next section for more details).

3. RELATED WORK

UniMod [4] is a tool for specifying the logic of Java programs using state machines. It provides its own UML model editor as a plugin for the Eclipse platform. It also comprises a visual debugger, a validator, a compiler, and an interpreter. However, the notation does not support some features we need, such as parameters in call events and the possibility of using custom names for events.

Hugo/RT [30] is a UML model tool that supports model checking (using the SPIN [2] model checker) and code generation (e.g. Java). In particular, given a set of state machines, that describe the behavior of different objects, and a collaboration diagram Hugo/RT can prove if the scenario described by the collaboration diagram can be generated by the execution of the state machines. Thus, Hugo/RT is more focussed on modelling and analyzing protocols rather than, as FSMC+ does, properties of the state machines.

Tabu [7] is a UML model checker based on SMV with emphasis on usability also by users with little or no knowledge

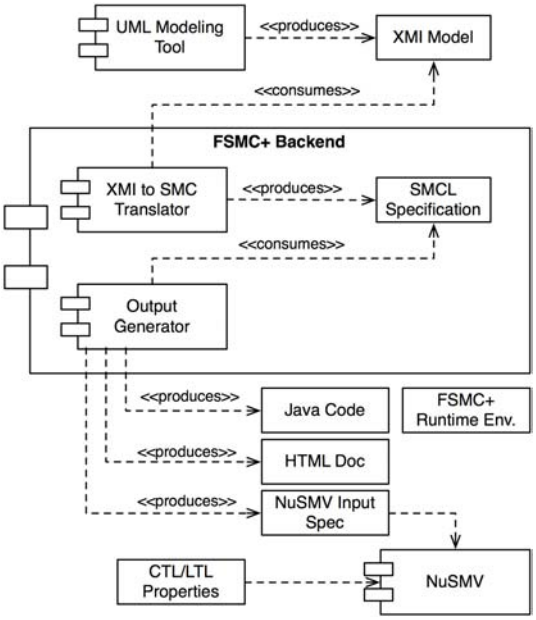

Figure 1: FSMC+ Architecture

in formal verification and temporal logic. Despite its appealing approach which comprises an *assistant* for guiding the user in the verification task, it lacks, to our knowledge, the possibility of generating Java code.

SMC (State Machine Compiler [29]) is an open-source translator from a textual representation of a state machine to various output formats (Java, C, C++, HTML, etc.). FSMC+ is partially based on SMC. FSMC+ maintains the internal textual representation for state machines defined by SMC and, partially, the structure of the generated Java code, along with the runtime library. The code generator, however, is completely rewritten and so is the compiler to the NuSMV language. Moreover, the module which implements the XMI-to-SMCL translation (see Section 4) is a completely original effort of our project.

Providing a detailed review of the work related to define a formal semantics for Statecharts and to compile them in a language suitable for formal verification is outside the scope of this paper. Suffice it here to mention that FSMC+ follows the work presented by Clarke and Heinle in [12].

4. THE FSMC+ TOOL

FSMC+ is composed of two main components, as shown in Figure 1:

- **a UML modeling tool.** The UML modeling tool is used as a graphical front end to write one or more Statechart(s), that specify the control logic of the system, that is, how the system interacts with its *environment* and the *logic* it must implement.
- **the FSMC+ Backend**. The FSMC+ backend translates the UML Statecharts into the target languages. At the moment, we support the following three outputs:

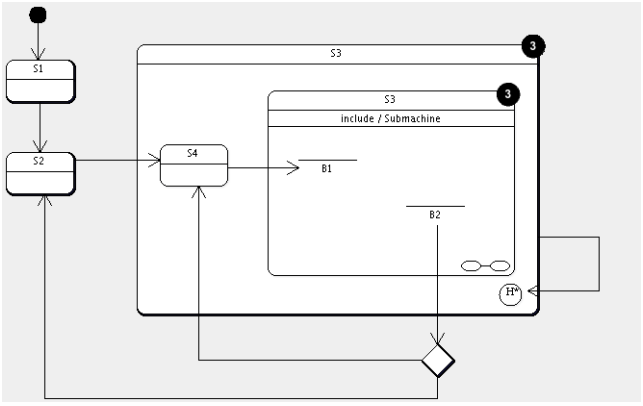

Figure 2: Supported Notation (I)

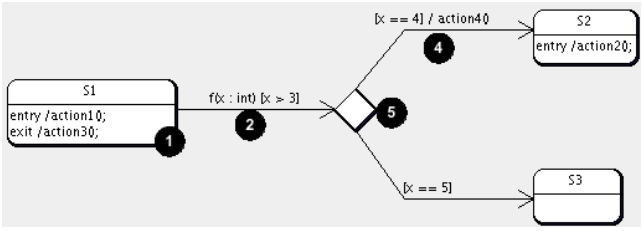

Figure 3: Supported Notation (II)

1. *Java code*, that, when put together with a set of run-time libraries, implements, in Java, the *control logic* specified by the Statecharts;
2. *NuSMV source code*, that can be given as input to the NuSMV model checker in order to verify whether the logic specified by the Statecharts satisfy a set of properties, written in CTL or LTL temporal logic;
3. *HTML documentation*, which is used for documentation purposes and for some inspection activities.

In the following subsections we provide a more detailed description of the different components of FSMC+ and some hints on its usage.

4.1 Step 1. Model the Control Logic

The first step when using FSMC+ is writing the control logic of the system using UML Statecharts.

FSMC+ is based on UML metamodel version 1.4 [5] and supports a subset of the UML Statechart notation, which has been specifically customized for Java generation. In particular (numbers in parenthesis refer to Figures 2 and 3):

- the states' entry and exit actions can be sequences of Java method invocations (1);
- transitions can use
  - guards that refer to event parameters and to the internal state of the machine (2);

  - sequences of Java method invocations as the actions to be performed when the transition is executed (4);
  - junction pseudo-state as source or target state-vertexes, to simplify the notation (5).
- both call-events and signal-events are supported;
- parameters can be specified for call-events (2);
- signals can be organized in hierarchies to allow prioritization of transitions;
- composite states and submachine states allows for specification of the logic (3);

Even though we use ArgoUML [31] as the front-end, FSMC+ can use any UML tool that supports Statecharts and XMI export, version 1.2 [25].

4.2 Step 2. Generate NuSMV, HTML, and Java Code

The second step is invoking the FSMC+ backend to generate the target outputs.

FSMC+ is integrated in Maven [1], a freely available build tool. The FSMC+ backend, thus can either be invoked from the command line or integrated in the build process by using some maven plugins we specifically developed[3].

The generation is the responsibility of the following two components:

- **the XMI to SMC Translator**, which takes as input the XMI produced at the previous step and generates the SMCL representation of the Statechart (SMCL is the SMC input language [29]);
- **the Output Generator**, which takes as input the SMCL representation of the Statechart and generates the output in the target languages.

Dividing the translation in two distinct steps and basing it on a textual format allowed us to achieve the following goals (the first is specifically related to ProVotE's time constraints, whereas the second has a wider scope):

- *incremental development*: the output generator is the core function of FSMC+. By starting from SMCL and SMC we managed to incorporate Statecharts early in the development of *jprovote* and move to more sophisticated source generation and functions as the development of *jprovote* progressed;
- *manual inspection*: by dividing the translation in smaller steps we mitigate risks related to bugs in the generators and allow for manual inspections of the generated outputs. (Manual inspection of the code generated by the translation has been performed for the *jprovote* system).

[3]To the reader familiar with maven, FSMC+ runs as a plugin in the *generate-sources* phase.

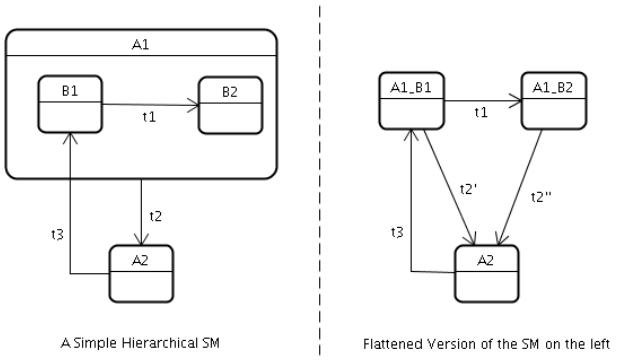

Figure 4: Flattening Example

4.2.1 From XMI to SMCL

The XMI-to-SMCL translator, developed in SWI-Prolog [3], reads an XMI specification produced by the UML modeling tool and transforms it into SMCL, a simple text-based representation of a Statechart.

The translation process is roughly divided in the following steps:

1. the XMI document is parsed using the SWI-Prolog's SGML package to produce a set of prolog facts that resemble content of the XMI;
2. from such set of facts an internal model for the state machine is deduced. The model is described by predicates whose names reflects the state machine domain i.e. *state machine, transition, action*, etc. The model is as rich as the UML one, i.e. it allows, for example, for composite states and concurrent regions;
3. the model obtained in the previous step goes through a *flattening* process, which basically removes composite and submachine states and leaves only simple states and (possibly inherited) transitions between simple states, maintaining the same semantics. (A simple example of flattening is drawn in Figure 4);
4. the simplified model is finally translated in the SMC representation, by means of a grammar specified as a DCG (Definite Clause Grammar).

Such an approach, in our opinion, has the following strengths:

- the generation of an internal model isolates tool's algorithms from external dependencies, e.g. UML metamodel and XMI version;
- the specification of the output format with a DCG grammar simplifies the extension of the tool to generate outputs in other modeling languages;
- the usage of a logic programming language significantly reduced the implementation effort of the flattening algorithm.

4.2.2 The Output Generator

The output generator takes as input an SMC specification and produces as outputs Java, NuSMV, and HTML. The generator is a completely rewritten version of the one provided with the SMC tool.

Code generation in FSMC+ is based on code templates and on Velocity [27], a template engine written in Java. For each target of the output generator we define, once and for all, a template that encodes the structure of the generated code. At run-time the structure is instantiated by Velocity taking as input the output produced at the previous step.

Using a template engine for specifying the target outputs allows, among other things, to easily extend the set of supported output formats and to change the style of the generated code.

4.3 Step 3. Verify the Control Logic

FSMC+ can translate the Statechart into the NuSMV input language. NuSMV [11] is a software tool for the formal verification of finite state systems. It has been developed jointly by ITC-IRST and by Carnegie Mellon University. NuSMV allows to check finite state systems against specifications written in temporal logics.

The third step, thus, consists in verifying the control logic with NuSMV.

The current implementation of FSMC+ implements a translation algorithm which resembles the one presented in [12]. As described in Figure 5, the code generated for the NuSMV verification consists of the following modules:

1. the state machine itself;
2. a skeleton of the *delegate modules*, that define the semantics of the actions invoked by the state machine;
3. a skeleton of *driver modules*, that model how events are generated.

To perform the verification the designer has to complete the skeletons of the delegate and of the driver modules. This corresponds to providing an implementation of the environment in which the state machine is run, namely, of the inputs the machine can receive (driver) and of the actions the machine performs on (delegate). For example, if the system being modeled is an interactive application provided with a graphical user interface, the driver will model the user actions and the delegate the changes in the interface required by the state machine as responses to the user actions.

The level of abstraction to keep in the specification of the environment is up to the user. The simplest implementation of the driver and delegate modules consists in an environment in which inputs are completely random. The random environment is the simplest model to produce with NuSMV, as it does not require to write a single line of code. Moreover it produces the harshest environment in which to test the state machine. In various real-life situations, however, designers may want to put some constraints on the inputs

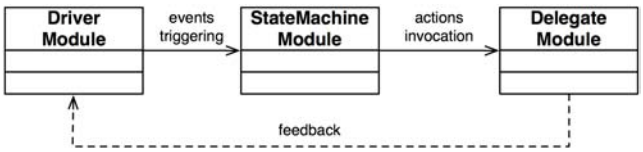

Figure 5: NuSMV Specification Structure

and/or provide some constraints between the outputs generated by the delegate module and the inputs produced by the driver module. Even though defining such constraints requires more work, it allows to tackle the state explosion problem, namely, keep the formal verification feasible.

After having modeled the system, the user has to define the properties the system is supposed to have. NuSMV allows properties to be specified either in CTL (Computation Tree Logic) or LTL (Linear Temporal Logic). Such properties usually involve delegates' state variables and prescribe intended system behavior deduced from system requirements traditionally classified in terms of *safety* and *liveness*.

In addition to the user-specified properties FSMC+ produces a set of properties aimed at verifying various syntactic characteristics of the model to ensure that

1. the designer correctly used the invocation model of the state machine on the delegate
2. the translation process from Statecharts to NuSMV was successful.

The last action consists in launching the NuSMV model checker, which reads the model, the properties and verifies whether the properties are always satisfied, that is, in any possible execution of the system.

If a property can be violated by the system NuSMV outputs a trace with the counterexample. By analyzing the traces produced by NuSMV the designer can find the conditions that cause the system to fail and fix the problem. The process is iterated until every property is satisfied.

4.4 Step 4. Integrate the Java Code

The last step consists in integrating the generated Java code in the system.

The structure of the Java code generated by the FSMC+ comprises (see Figure 6):

1. a class which implements the logic described by the Statechart (*StateMachine* in the figure)
2. an interface which collects all the Java methods and boolean conditions mentioned in the Statechart (*DelegateInterface* in the figure).

The *StateMachine* class exposes a method for each call event in the Statechart. For example the call event $f(i : int)$

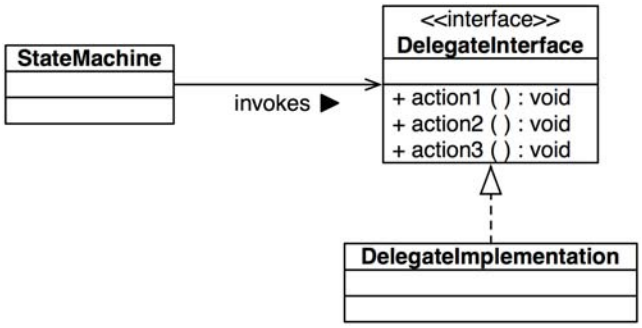

Figure 6: Generated Code Structure

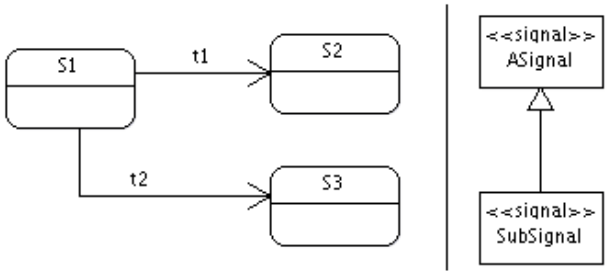

Figure 7: Signal Handling

defined in Figure 3 tells FSMC+ to define a method of the same signature.

The interface generated by FSMC+ has to be implemented by an application class (*DelegateImplementation* in the figure) to specify the proper semantics of actions and conditions.

FSMC+ supports two strategies to effectively integrate a FSMC+ generated state machine into a system and designers are left to choose, depending on the situation, which one is better.

The first strategy consists in providing a class that dispatches the events to the state machine, explicitly triggering its execution.

As an alternative modeling strategy, FSMC+ allows to use a signal-oriented paradigm for modeling transitions. In this case the state machine reacts to *signal events* which are, according to UML ver. 1.4, occurrences of *Signals* and that FSMC+ requires to be organized in a hierarchy, i.e. every *Signal* must derive from a root *Signal*. To support such an approach FSMC+ generates the state machine class having just a *handle(TopSignal s)* method, supposing that, for example, *TopSignal* is the root signal which is the parent of every signals in the model. To solve conflicts in selecting which transition to fire, FSMC+ follows the signals' hierarchy. For example, as shown in the left side of Figure 7, if the state machine contains two transitions with trigger *t1* (occurrence of *ASignal* not shown in the figure) and *t2* (occurence of *SubSignal* not shown in the figure) *t2* will be fired because *SubSignal* is more specific than *ASignal*.

5. CASE STUDY: THE JPROVOTE SYSTEM

In the following sections we provide some hints related to the application of FSMC+ to the development of the *jprovote* system.

We start by providing some insights related to the adoption of e-voting system in Italy, continue by describing the e-Voting machine we developed, and describing how we applied FSMC+ for specifying and verifying the control logic of the e-voting machine.

5.1 Voting and e-Voting in Italy

The current Italian ballot system consists of a paper-and-pencil method and electors are allowed to vote only in the polling station where they are registered. Simplifying both on the law and on the procedures for the sake of presentation, voting in Italy happens as follows:

1. **identification and registration of the voter.** At the polling station the voter is usually required to show his/her ID card and the electoral card. If the name of the voter is present in the electoral list of the polling station, the voter is registered, the electoral card stamped, and the voter is admitted to voting.
2. **casting a vote.** The voter is given a ballot and a pencil and is shown a cabin where the vote can be cast in secrecy. Secrecy is both a right and a duty. The Italian law and procedures are aimed at ensuring that the voter cannot make his/her vote manifest to other people.

At the end of the voting day, the ballot boxes are opened and the counting procedure starts:

3. **counting.** Votes are counted and the results tabulated in special registers. The Italian law aims at protecting the intention of the voter. Thus, even if a vote is not compliant with the definition given by the law, the vote may still be partially assigned and counted, if some its parts can be unambiguously interpreted. Representatives of the political parties monitor the counting procedure.
4. **transmission of the results.** When all the ballots have been tabulated, the results are transcribed in various paper documents and transmitted to the offices responsible of aggregating all the data.
5. **sum and proclamation of the elected representatives.** All the data coming from the different polling stations are counted and seats assigned according to algorithm defined by the law. Data are then made available to the general public.

Systems for automating steps 4 and 5 above have been in use, for some time, in Italy. Even though they are used only to make provisional results available, the verification procedures generally confirm the provisional results and voters usually looks at the results produced by these systems are the "official" ones.

Figure 8: The ProVotE Machine

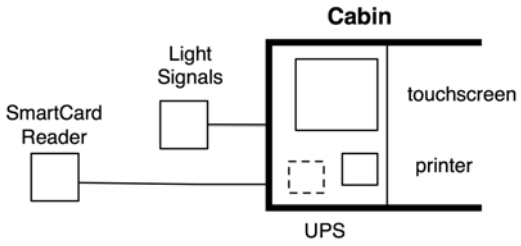

Figure 9: The ProVotE Machine

Some experimentations have been conducted in Italy to try and automate the other phases as well. The largest trial, so far, was sponsored by the central government, and concerned a system for automating steps 3 and 4 above. The system, operated by specially appointed technicians, was installed in 47 precincts at the last European elections and repeated at the last political elections (2006). Little, however, is known about the results of the experimentation. See [18] for some more details.

Proper e-voting experimentations (i.e. including step 2) have been conducted at the local level, usually on a small scale, in experimentations which seem to have had little continuity and/or on which information is scarce. We mention San Benedetto del Tronto (2000), trials sites in Avellino (2001), Campobasso (2001), Cremona (2002, 2006), Ladispoli (2004), Specchia (2005) [14, 16]. Other experimentations have been conducted in Valle D'Aosta, Friuli Venezia Giulia, and Milan.

Independently from their scope, past experiences demonstrate the extreme caution of the central and local government in trying and modernizing the way in which voters express their vote and the difficulty in overcoming doubts and suspicion in citizens. Providing usable, reliable, safe, and secure systems also for trials becomes extremely important, as trust on the systems is built also during experimentations. Proper development of e-voting prototypes, therefore, not only plays a key role in contributing to the quality of the final product, but it is also an essential step for project success.

The experimentations conducted within the ProVotE project are among the biggest (if not the biggest) experimentation of e-voting in Italy.

5.2 The ProVotE e-voting System

Even though the solution we developed in trials involves various subsystems (to cover steps 1-5), the e-voting machine is probably the most critical and "visible" component.

In fact, the e-voting machine has to be continuously available during the electoral day (6.00am to 10.00pm in Italy), it has to store votes safely and securely, it is used by all the voters and, as mentioned earlier, it is one of the main systems on which citizens judge the feasibility of e-voting.

The ProVotE e-voting machine we developed (see Figure 8) is a DRE with a voter verifiable printed trail as suggested by e.g. [23, 24] with a signaling system to specify the status of the machine and a smartcard reader to enable operations on the machine.

The voting machine is installed in a standard voting cabin as shown in Figure 9.

All the interaction with the voting machine happens through a touchscreeen which, during voting, reproduces the paper ballot. Votes are printed and shown to the voters, who are required to confirm them or make them void. Verification by the voter increases auditability of the system. The printer tape is cut after each vote (and the electronic data is *shuffled* after each vote), so that no sequence remains[4].

The machine is locked by default. A smartcard reader, outside the cabin, is used to unlock it. Two smartcards, that need to be alternated, are used to enable the machine for voting. This helps avoiding the possibility of casting more than one vote per voter. A third smartcard is used for the functions of the precinct's officers (e.g. opening and closing elections).

A set of lights (two green and one red) is used to inform about the state of voting machine. The red light signals that the machine is being used for voting and the green ones encode which card will enable the machine for the next vote.

The voter can abort the voting operation at any moment (for instance, if in difficulty) by extracting the smartcard. In such a scenario the machine has to be reset so that no trace of the voter's intent remains revealed.

The electronic ballot data are stored by means of redundant hardware: all the data generated by the machine is written both in the internal hard-disk and in a removable mass storage device. This solution allows for the detection of hardware failures in the storage devices. Electronic ballots are scrambled, encrypted (during the voting day) and signed (after the voting day).

The software system is based on a stripped-down version of Linux and an application (*jprovote*) written in Java (following e.g. [21]). The *jprovote* system is structured in four layers:

- **services,** which provides the basic functionality to the

[4]Being able to determine the order in which votes are given has resulted to be one of the concerns of Trentino's voters.

rest of the application, such as drivers for controlling the three-light indicator and the printer, managing logs for audits, and transparently managing redundant and ciphered persistence of data;

- **data model management,** which manages all the election specific data, comprising candidates and parties, the ballot data, per-machine election results, and the symmetric and asymmetric keys used for ciphering and signing;
- **control logic,** which defines how the machine has to react to user actions both in administration and in voting mode. The control logic also specifies the logic of the user interface (e.g. what screens has to be shown next).
- **user interface,** which manages the graphical layout of the administration and of the voting interface.

The control logic component of *jprovote* is among the most critical part of the system. The component, in fact, is responsible of implementing a well-defined life cycle, which encodes the procedures defined by the law and which limits the operations poll-officers can do.

Errors in its specification may lead to violations of fundamental principles on which the law is based (e.g. allowing a voter to vote twice, denying the right to vote to a voter). Finally, the e-voting machines need to be adaptable to different kind of elections and come in different configurations (e.g. testing, voting), which require both flexibility in the specification of the user interface and efficiency in being able to adapt the behavior of the control logic. We decided, therefore, to specify the control logic using finite state machines.

5.3 Control Logic Development

Broadly speaking, the process we followed for the development of the e-voting system is a waterfall which we adapted to incorporate (formal) verification activities. Not surprisingly (as the system was developed for the development of the e-voting system, as mentioned earlier), FSMC+ significantly helped during the development phases.

The process is summarized in Figure 10:

Process Modeling an initial set of discussions and analysis of the Italian electoral law allowed to provide a formalization and a view of the voting procedures in Italy. The formalization was given using activity diagrams in UML and textual descriptions. Such documentation served as the basis for the subsequent requirements analysis.

Requirements Definition the initial requirements definition activity produced a requirements document which contains high-level Statecharts and use cases. The Statechart specifies the life cycle of the machine (e.g. opening the poll-site, starting the counting, etc.), according to the processes identified above and the use cases provide the usage scenarios for each of the states of the machine's life-cycle (e.g. the actions necessary to open the poll-site).

The voting and the administration user interfaces are specified through a Statecharts which describes their logic and

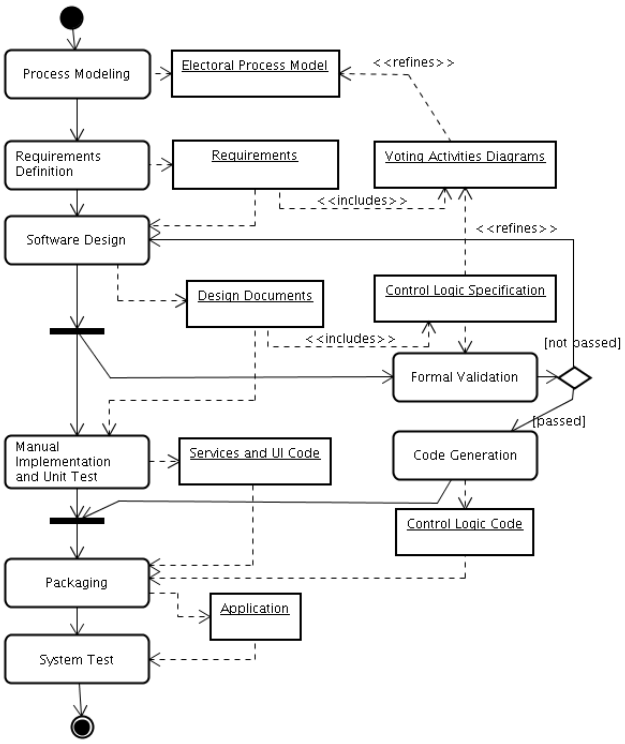

Figure 10: ProVotE Development Process

in which each state corresponds to a different "screenshots" of the system.

The specifications and, in particular, the UML Statecharts have been discussed and validated by the Electoral Service of the Province of Trento. The voting interface was validated using throw-away prototypes developed from the Statecharts, in experiments set up by the Faculty of Sociology with selected voters before the trials.

Software Design During the design phase, the Statecharts validated by the Electoral Service have been detailed into an *executable* Statecharts, that is, Statecharts that we can give as input to FSMC+ to generate code that controls the e-voting machine.

Formal Validation The *executable* Statecharts specification have then been translated using FSMC+ into the NuSMV input language and formally verified.

For the validation we started by providing simple NuSMV models for the environment of the control logic and manually specified the main principles/properties that the machine had to satisfy using CTL.

In the model of the environment, for example, we specified the semantics of actions that controls the state for hardware parts such as the lights (e.g. which light is on), the printer (e.g wether a vote is exposed in the printer or not), the screen (e.g. which form is shown to the user), the internal state of the memory, etc. We also model the behavior of input hardware, i.e. the smartcard reader and the touch-

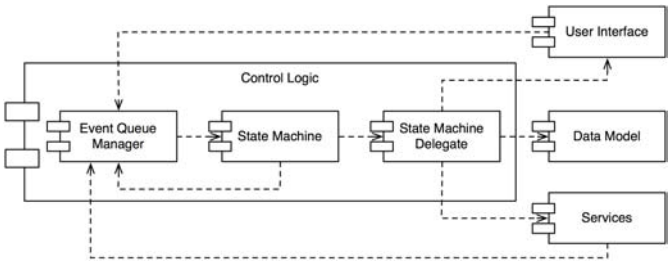

Figure 11: JProvote Application Architecture

screen. Furthermore, an example of the feedback loop cited above is the dependency from the state of the screen on the type of signals that could be generated, i.e. the user could confirm the vote only if the button to do it is shown on the screen.

The CTL properties are mainly expressed in terms of actions the control logic has to perform on the peripherals to ensure that no basic principle of the law is violated (e.g. the machine is always put in a safe state if the current voter has to be interrupted) and that the behavior of the machine is compliant with the specification provided in the requirements document (e.g. the driving of the signaling and log system comply with the specification provided in the requirements document).

Interestingly enough, even the violation of trivial properties, such as the last one, are critical. Matter of fact, if the system does not properly log and signals whether a voter has cast his/her vote, the voter may be procedurally (i.e. by the polling officers) negated the right to use the system to express his/her opinion or given the possibility to vote twice.

See, e.g., [22] for a list of the properties verified. Notice that in order to standardize the representation of the properties, we used the patterns found [15].

Code Generation After having validated the specification of the control logic, we used FSMC+ to generate the Java code of the *control logic* and of the *user interface*.

The architecture of the software is shown in Figure 11.

An *Event Queue Manager* handles all asynchronous events (e.g. smart-card inserted/extracted events, power fail events from the UPS) and feeds the *State Machine*, which reacts to the events and sends commands to a *Delegate*. The *Delegate*'s methods can be thought like atomic building blocks (e.g. 'print the current ballot', 'clear the interface', 'turn the outside red light on', etc.) that are combined by means of the Statechart specification.

The Java code generated by FSMC+ was inspected by hand, to mitigate risks related to bugs in the translator, and the source code then compiled and packaged to produce *jprovote*.

Manual Implementation and Unit test The code implementing the *services*, *data model management* layers and some glue code to link the user interface was implemented and tested using standard practices.

Packaging and System Test The code generated by hand and that produced by FSMC+ have been then packaged together.

The formal verification performed with FSMC+ does not eliminate the need for testing, as properties related to e.g. integration of the different components, actual implementation of the drivers (is the driver really turning on the green light, when the system tells to do so?), etc. need to be performed on the actual system.

Thus, the application was finally verified during the System Test phase, conducted using standard techniques.

Even though the process followed above does not guarantee that the system is error-free (due to, e.g., completeness of properties, errors in the translators), the use of graphical notations and the formal verification of the control logic allowed to significantly increase our confidence in the conceptual correctness of the design of the control logic. The use of Statecharts, in fact, significantly simplified interaction with other stakeholders and increased understandability of the requirements by the people in the electoral service. (The Electoral Service is now evaluating the possibility of making the activity diagrams used to specify the electoral processes available to the general public.)

6. CONCLUSIONS AND FUTURE WORK

This paper presented FSMC+, a tool for automatically generating Java code and NuSMV specification from UML Statecharts. FSMC+ can simplify the development of complex/embedded systems by supporting a model-driven approach in which certain components are automatically generated from high-level specifications.

Various tools, both commercial and freely available, exist to support the approach FSMC+ adopts. However, some of the features FSMC+ provides are peculiar and/or not readily available in other similar tools, among which a rather complete support to the UML ver. 1.4 Statechart notation and a flexible and extensible architecture.

Even though FSMC+ has been specifically created to ease the development of *jprovote* we believe the approach and the tool we developed to be general enough to be used in other applications.

Future work will include the extension of supported notation to concurrent regions and to new versions of the UML metamodel, which are not currently supported.

The *jprovote* system has been used, so far, in four different experimentations held during local elections and for electing the representatives of the students in a local High School. So far more that 11000 citizens tried our system. No glitches have been signaled during the experimentations.

7. ACKNOWLEDGMENTS

The work described in this paper has been and is being developed within the ProVotE Project, a project sponsored by Provincia autonoma di Trento.

As with any large project, the results presented in this paper

based on the joint and coordinated work of several people. We wish in particular to thank: S. Bettotti, C. Buzzi, L. Caporusso, A. Ceschi, C. Charalabopoulos, G. Conti, C. Covelli, R. Deamicis, G. Fasanelli, A. Faustini, G. Fele, P. Gentile, L. Giuliani, F. Gleria, B. Lanzalone, A. Lavarian, G. Negri, E. Parola, E. Passante, G. Pedrotti, P. Peri, N. Prantil, R. Resoli, F. Sartori, G. Stellucci, C. Tonini, P. Troebinger, M. Welponer.

8. REFERENCES

[1] Apache maven project. `http://maven.apache.org`.

[2] On-the-fly, ltl model checking with spin. Available at `http://spinroot.com/spin/whatispin.html`.

[3] Swi-prolog's home. Available at `http://www.swi-prolog.org/`.

[4] Unimod. Available at `http://unimod.sourceforge.net`.

[5] Omg unified modeling language specification, 2001. version 1.4.

[6] B. V. Acker. Remote e-voting and coercion: a risk-assessment model and solutions. In *the International Workshop on Electronic Voting in Europe*, 2004.

[7] M. E. Beato, M. Barrio-Solrzano, and C. E. Cuesta. Uml automatic verification tool (tabu). In *SAVCBS'04 Specification and Verification of Component-Based Systems Workshop at ACM SIGSOFT 2004/FSE-12*, 2004.

[8] M. Book, G. Beydeda, and V. Gruhn. *Model-driven Software Development*. Springer Verlag, 2005.

[9] L. Caporusso, C. Buzzi, G. Fele, P. Peri, and F. Sartori. Transitioning to evoting and citizen participation. In *Proceedings of eVoting-2006*, 2006.

[10] D. Chaum. Secret-ballot receipts: True voter-verifiable elections. In *CryptoBytes*, volume 7 (2). RSA Laboratories, 2004.

[11] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[12] E. Clarke and W. Heinle. Modular translation of statecharts to smv, 2000.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[14] M. I. Comune di San Benedetto del Tronto. Preliminar report on the electronic voting expreimentation (sperimentazione sulla votazione elettronica - preliminare). In Italian. Available at `http://www.comune.san-benedetto-del-tronto.ap.it/ePoll/rl00.html`.

[15] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. Technical Report UM-CS-1998-035, , 1998.

[16] E-Poll. Electronic polling system for remote voting operations. Available at `http://www.e-poll-project.net/`.

[17] H. Goldsby, B. H. Cheng, S. Konrad, and S. Kamdoum. A visualization framework for the modeling and formal analysis of high assurance systems. pages 707–721, 2006.

[18] Governo Italiano. *European Elections 2004, Automated Counting of the Votes (Elezioni Europee 2004, Conteggio Automatizzato del Voto)*. In Italian. Available at `http://www.governo.it/GovernoInforma/Dossier/voto_conteggio`.

[19] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[20] J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer Magazine*, 23(9):8–24, Sept. 1990.

[21] T. Kohno, A. Stubblefield, A. Rubin, and D. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, 2004.

[22] M. McGaley and J. Gibson. Electronic voting: A safety critical system. Final year project report, 2003.

[23] R. Mercuri. Explanation of voter-verified ballot systems. *ACM Software Engineering Notes (SIGSOFT)*, 27(5). Also at `http://catless.ncl.ac.uk/Risks/22.17.html`.

[24] R. Mercuri. A better ballot box? *IEEE Spectrum Online*, October 2002. Available on line at `http://www.spectrum.ieee.org/WEBONLY/publicfeature/oct02/evot.html`.

[25] OMG. Omg xml metadata interchange (xmi) specification, version 1.2, January 2002.

[26] A. Ostveen and P. van den Besselaar. Security as belief - user's perceptions on the security of electronic voting systems. In *the International Workshop on Electronic Voting in Europe*, 2004.

[27] T. A. J. Project. Velocity. Available at `http://jakarta.apache.org/velocity`.

[28] A. Prosser, R. Kofler, R. Krimmer, and M. K. Unger. Security assets in e-voting. In *the International Workshop on Electronic Voting in Europe*, 2004.

[29] C. W. Rapp. The state machine compiler. http://smc.sourceforge.net/.

[30] T. Schafer, A. Knapp, and S. Merz. Model checking uml state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55, 2001.

[31] T. O. S. S. E. Tools. Argouml. Available at `http://argouml.tigris.org/`.

[32] A. Villafiorita and G. Fasanelli. Transitioning to evoting: the provote project and trentino's experience. In *Proceedings of EGOV-06*, 2006.