Statecharts composition to model topologically distributed applications

Michele Banci
Istituto di Scienze e Tecnologie dell'Informazione, CNR
Pisa, Italy
Alessandro Fantechi
Dipartimento di Sistemi e Informatica, Università di Firenze
Firenze, Italy
Stefania Gnesi
Istituto di Scienze e Tecnologie dell'Informazione, CNR
Pisa, Italy

ABSTRACT: Many real-life systems can be conveniently modeled by the replication and interconnection of simple components of a few types: different configurations of the same system may vary just for the number of components and for the topology of their interconnections. In industrial practice of formal modeling the tedious work of such a manual instantiation has to be automated, which allows to produce at a low cost new configurations of the same product.
This paper presents a methodology to build statechart models of topologically distributed systems by instantiating generic components; the method is able to replicate as needed statechart components and to expand the terms that drive their behavior to create complex interconnection patterns along the required topology. The methodology is illustrated on the example of a classical distributed algorithm (Byzantine Agreement) to show the potential of dealing with parameterized distributed structures. The proposed methodology is shown to be able to find several interesting industrial applications.

Keywords: complex systems, instantiation, formal specification, statecharts, state diagrams, component based systems.

I. INTRODUCTION

Formal modeling of complex systems is more and more adopted as a way to anticipate, by simulation or formal verification, the problems that may be faced by the actual implementation. Many complex real-life systems can be conveniently modeled by the replication and interconnection of simple components of a few types: different configurations of the same system may vary just for the number of components and for the topology of their interconnections.

In industrial practice of formal modeling the tedious work of such a manual instantiation has to be automated, which allows to produce at a low cost new configurations of the same product. This paper presents a methodology to build statechart models of topologically distributed systems by statically instantiating generic components; the method is able to replicate as needed statechart components and to expand the terms that drive their behavior to create complex interconnection patterns along the required topology.
The methodology we propose is particularly suited to be applied when modelling a *family* of different applications, which differ among them only for the configuration of components, both in number and interconnection, often corresponding to different topological distributions of components of a modelled physical system.

If we adopt the terminology used in Product Family engineering [4], [7], the development of a product family is characterized by two processes: *domain engineering* and *application engineering.* Domain engineering is the process aiming at developing the general concept of a product line together with all the assets which are common to the whole product line, whereas application engineering is the process aiming at designing a specific product.

In the framework we propose, domain engineering produces a set of generic statecharts, which constitute the basic components of the architecture of any single product of the family. The generic statecharts comes accompanied by a set of general instantiation rules which define how to replicate, parameterize and interconnect the components to form the architecture of a single product complying with the family.

Application engineering becomes then an instantiation process, which is driven by a database containing the knowledge about the (often topological) configuration of the architecture of the single product of the family which is actually built. This instantiation process is therefore conducted by means of queries to such a database, following the general instantiation rules defined for the family.

Usually, products belonging to a family differ among them for a limited number of discrete parameters, or for the optional presence of some functionalities (often called *features*). Therefore, the complexity of the application lies mostly in the complexity of the provided features, and far less in their configuration and parameterization.

We address instead product families where most of the complexity lies in the configuration, while basic generic el-

ements can have very simple behaviour and functionality. In particular, the classes of systems that we address share the possibility to be formalized as follows: a particular system of the family is modeled composing patterns of (simple) statecharts in static configurations. Composition rules actually depend on the family (domain engineering) . We do not stick to any particular dialect of statecharts, hence interactions may occur either via events, common variables, common objects, or remote method calls, whatever is appropriate for the formalism at hand (i.e. Statemate or Stateflow statecharts, UML statecharts,...). In fact the methodology acts at the syntax level, and therefore it does not introduce any sort of restriction with regard to the semantics used for the model.

We exemplify the approach over a distributed algorithm (Interactive Consistency), built by reusing the same basic components.

This approach may find useful applications in the modelling of several classes of statically configured distributed systems; in particular we refer here to our experiences on railway signalling systems [1], but also to systems controlling other kinds of physical networks.

This paper is structured as follows: in section II we remind some useful notions about statecharts. Section III discusses the proposed instantiation process over generic statecharts, while section IV describes the case study and the proposed process applied to it.

II. Harel Statecharts, UML State Diagrams, Stateflow Diagrams

Harel statecharts [5] formalism is an extension of classic formalism of Finite State Machines (FSM), to allow hierarchical parallel interacting state machines to be specified.

One of the main notions of statecharts are that of states and transitions among them. A transition connects a *source* to a *target* state. The transitions are labelled by a trigger event, a boolean guard and a sequence of actions.

"System states" are modelled by *configurations*, which are sets of states. A transition is *enabled* and can fire if and only if its source state is in the current configuration, and the guard is satisfied. In this case, if the transition fires, the source state is left, the actions are executed, and the target state is entered.

In the general case, some target sub-states can be explicitly specified, and more than one event can be available in the environment. UML Statechart Diagrams [10] are a (object-oriented) variant of classical Harel statecharts and the UML semantics assumes a *dispatcher* which selects one event at a time from the environment, modelled as a queue, and offers it to the state machine.

When the effects of all such transitions and related actions are complete a new event is selected by the dispatcher and a new cycle is started. In this sense the UML semantics does not allow "chain reactions" within the same step: events generated as a consequence of firing a step are not available to the machine during the same step, but they are available for being dispatched to the machine only from the next step on.

Several dialects of Statecharts are actually used: UML State Diagrams and Statemate Statecharts [5] are currently the most popular ones, but other ones like Stateflow [8] have a large use in several industrial applications.

The statecharts dialects differ among them mainly because:
1. they give a different semantic interpretation to statecharts, mainly with respect to priority of transitions firing and nondeterminism resolution;
2. they adopt different ways to structure a complex system in a set of statecharts.

III. The proposed methodology

The methodology we present consists of a formal framework in which developing models of complex systems that belongs to the same family of applications.

Sometimes in industrial applications there is the need to build systems in several different configurations: these configurations are distinct systems that actually have a common core. An example can be found in the field of computer-based controllers of railway yards, where we can identify distinct track layouts, which differ each other by their configuration, but the related controllers act using the same basic control rules: this is what is often called a *system family*.

So, instead to redesign a new system every time, the main idea is to use a template to easily configure a system in order to design each new system.

The objectives are both human effort reduction in redesigning "similar" systems and reduction of human errors because of the use of the same generic model: in this case the confidence becomes higher at every new design of systems of the same family.

The proposed methodology is based on the definition of generalized objects. In these objects all the possible configurations have to be embedded.

The objects required by the methodology are:
- a set of generic statecharts, in which basic control rules about physical and logical components are embedded (see Sec .III-A);
- a set of instantiation rules (see Sec .III-B);
- a set of expansion rules (see Sec. III-C).

The proposed modeling process consists in the use of instances of the generic statecharts to model identical components of the modeled system. The instantiation process follows two steps: charts instantiation and terms expansion.

Our instantiation approach is actually independent by the particular dialect of statecharts, since on one hand it operates at the syntactic level, so ignoring semantic differences, while on the other, we assume just a flat structure of peer statecharts, so avoiding dialect-dependent architectural features.

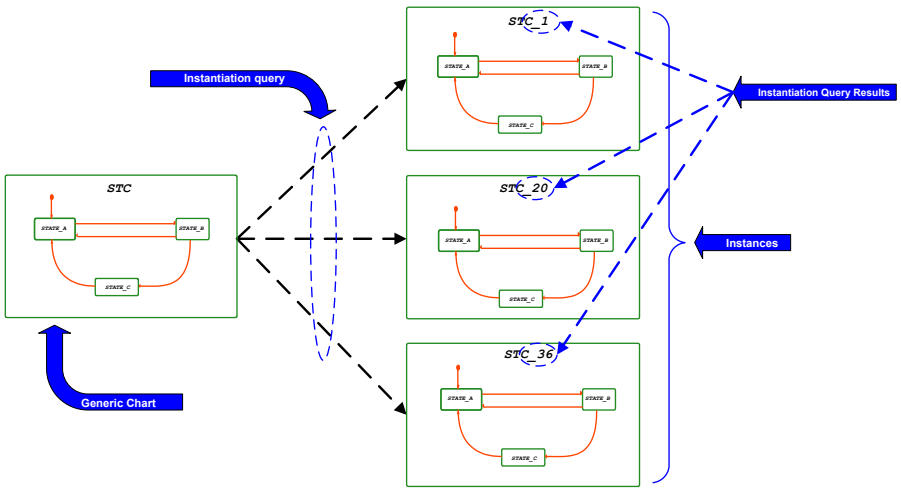

Fig. 1. Static instantiation process

A. Generic charts and instantiation process

The main concept on which the framework is based is that of generic chart. A generic chart is a template, in which the structure of states is fixed and the transitions can be parameterized. A generic chart is intended to be instantiated by proper actualization of parameters.

Furthermore, in our case in addition to having formal/actual parameters we introduce also a further level of genericity by means of generic terms (see Section III-C).

In the left part of Figure 1 an example of a generic chart is shown: these kind of charts are representative of the basic behavior of components. The distinctive features of a generic chart, which are not immediately apparent from the figure, are that:

- a generic statechart is a template, rather than the model of an actual element, hence its name is a generic name;
- a regular statechart defines names for variables, events, objects, visible to other statecharts; in a generic statechart, these, referred in the following as *contextual symbols*, have generic names;
- a regular statechart uses names of variables, events, objects, defined by other statecharts; in a generic chart, *generic terms* are used instead, which are placeholders to be expanded with (possibly complex) expressions over variables, events, objects defined by other statecharts. Generic terms provide another dimension of genericity to generic statecharts, that, since expressions on the transitions define the interconnections among statecharts, will be modified in relation to the specific application topology.

B. Chart instantiation

Chart instantiation consists in first replicating, adding suffixes, each generic chart as many times as needed: usually this *replication* serves to model the presence of a multiplicity of identical components in the modeled system. Each replica is given a unique name by adding a proper suffix. The contextual symbols need to be replicated together with the chart, and hence the same suffix is appended to their names (*renaming*). renaming does not affect non-contextual symbols and generic terms.

This instantiation is driven by an application database, which is given as input to the instantiation process, and which collects all the conditions and interrelations that have to be satisfied by the application. In general it is the application database that embeds the knowledge about the specific control rules for the developed system. In this way the instantiated charts maintain the same basic state behavior of the original one, but differ for the modality to reach the states.

Figure 1 shows the instantiation process driven by the instantiation query executed on the application database, and the suffixes added to the instantiated statecharts.

Hence, the definition of a generic chart is completed by an *instantiation query* which is used to extract the set of instance suffixes from the database, so that the replication and renaming are driven by this query. For instance, according to the structure of the application database, we can use a standard SQL language to formalize the query, as exemplified here by:

```
SELECT instance_suffix FROM a_table
WHERE something.
```

Anyway, its complexity depends solely by the adopted formal structure of the database.

At this step the instances have got the same state structure and the same transitions of the generic charts, furthermore the contextual symbols have been renamed (see Fig. 2) adding suffixes provided by the instantiation query results.

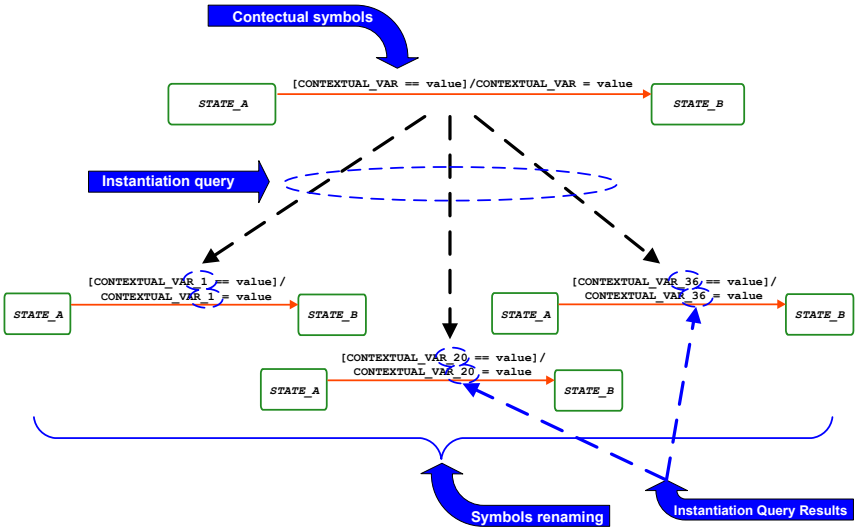

Fig. 2. Symbols renaming

Other non-contextual symbols will be left unchanged; these could belong to either "generic term symbols" or "global single symbols" which are not interested by the instantiation and expansion process, these represent variables not directly related to a particular instance but to the whole application.

C. Terms expansion

While instantiation (replication and renaming) rules produce effects on the number of charts and on some variable parameters, the *term expansion* solves the other dimension of genericity, conveying the information that is missing to relate generic terms to the actual application topology. A term expansion is a formalized rule that drives the construction of an expression related to the specific topology from a generic term. An expansion rule is based on one or more queries to the application database, depending on its complexity.

Figure 3 shows the idea on which the expansion is based. There are represented only two instances, the term expansion rule is the same for a generic term but the queries representing it are different from an instance to another, in fact the suffix of an instance is passed to the term expansion query as a parameter. Basing on this parameter the term expansion query provides different results and the terms will be substituted by different variable names. The example represented in figure 3 is the following:

Using instance number=1:
$GENERIC\_TERM \rightarrow$
$[VAR\_7 == value] and [VAR\_8 == value] and$
$[VAR\_11 == value]$

Using instance number=20:
$GENERIC\_TERM \rightarrow$
$[VAR\_2 == value] and [VAR\_8 == value]$

IV. A simple Case Study: the Interactive Consistency protocol

Interactive consistency [2] focuses on the problem of reaching agreement among multiple processors (nodes) in presence of faults. The main difficulty to be overcome in achieving interactive consistency is the possibility of conflicting values sent by faulty processors: such a processor may provide one value to a second processor, but a different value to a third one, thereby making difficult for the recipients to agree on a common value. Interactive consistency solves this problem by using several rounds of messages exchange during which processor p tells processor q what value it has received from processor r and so on. This problem is also known as the Byzantine Generals problem [6] in the literature. Interactive Consistency is a generalization of the Byzantine Agreement protocol where each node sends its private value to every other node.
In this section we discuss, in details, an example of the interactive consistency protocol modeled by statecharts; we show a model representing a system composed by three nodes interconnected by a network with proprietary links between couples of nodes ($p \rightarrow q, p \neq q$). Then we will consider a generic solution, that can admit a number of n nodes, with $n \geq 3$. Obviously, the complexity of such a system increases rapidly with the number of nodes (e.g. the three nodes model requires 36 statechart instances).
For simplicity, we have however kept fixed the number of retransmission rounds to two (following [3], $r = 1$): this is another dimension of genericity which could well be dealt with by our approach.

Moreover, since we are here interested to show concisely

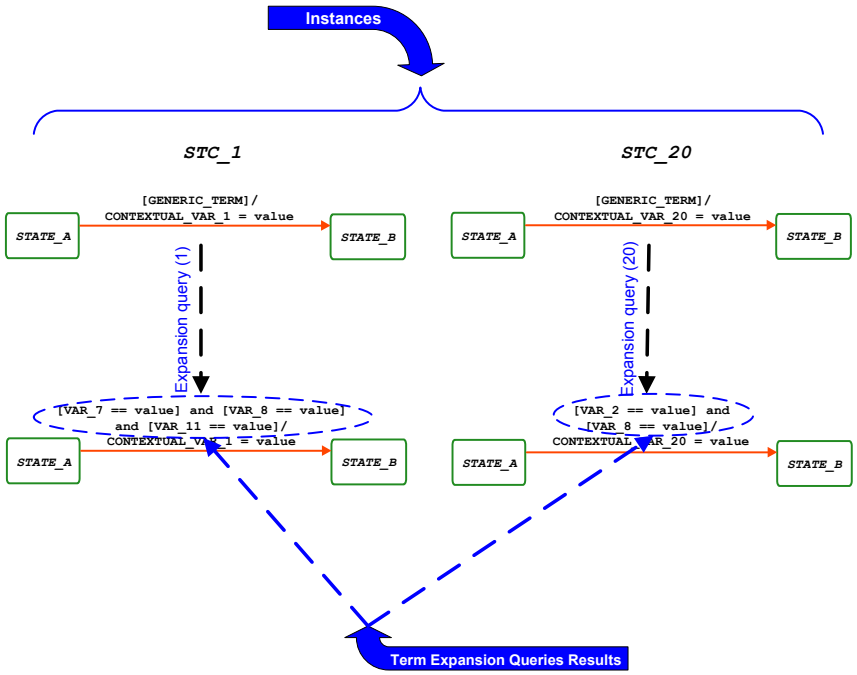

Fig. 3. Term expansion

how our instantiation process works, to limit the complexity of the example we consider only the fault-free behaviour. We have ignored here the modeling of faults, which is anyway central for the working of the Interactive Consistency protocol, since it has to guarantee given Validity and Agreement properties even in presence of faults (see [2], [3]). Actually, modelling faults is just a matter of added complexity which can be dealt as well in our framework.

A. Principle of the protocol

The protocol is implemented transforming the recursion which is typical of the Byzantine Agreement algorithm in a sequence of phases in which a node can send one message to every other node and simultaneously receive $n-1$ messages (one from each remote node). This solution is based on the concatenation of n BA phases [2].

Let n be the number of nodes and let the number of rounds be fixed at two, each node first sends its private value (round 1) and then successively relays the private values it receives from other nodes in a circular fashion (round 2). Making use of the symmetry of the problem just $n+1$ phases are needed on each node to implement the protocol. The $n+1$ phases are distinguished in: a first broadcast phase, $n-1$ broadcasted reception phases and a final voting phase.

S_phase_*1*	each node broadcasts its private value and receives (estimation) the values sent by the others $n-1$ nodes.
C_phase_*i*, for $i = 2..n$	one of the received message is relayed broadcasting it to the other $n-1$ nodes and it also receives (estimation) the values sent by the others $n-1$ nodes.
V_phase_*n+1*	all received message are majority voted by each node.

Having given the basic idea on which the algorithm is built, and having made some assumption (e.g. the fixed number of rounds), we have in this way identified a family of algorithms which differ each other by the number of nodes and consequently by the number of phases: we are hence able to build a generic model for this family, and some rules to automatically generate new algorithms with a reduced effort.

Table I shows an example for three nodes where each node sends its private value to every other node and during the final phase a voting about the received values is done. Each row of the table corresponds to a phase of message exchanges, within each phases there are three subphases (one broadcast subphase and two receiving subphases). The receiving subphases act concurrently inside each node.

		NODE P	NODE Q	NODE R
S_PHASE_1	TX_1	Tx p	Tx q	Tx r
	RX_1	q1:=Rx V_p(q)	r1:=Rx V_q(r)	p1:=Rx V_r(p)
		r1:=Rx V_p(r)	p1:=Rx V_q(p)	q1:=Rx V_r(q)
C_PHASE_2	TX_2	Tx V_p(q)	Tx V_q(r)	Tx V_r(p)
	RX_2	r2:=Rx V_p(V_q(r))	p2:=Rx V_q(V_r(p))	q2:=Rx V_r(V_p(q))
		p1:=Rx V_p(V_r(p))	q1:=Rx V_q(V_p(q))	r1:=Rx V_r(V_q(r))
C_PHASE_3	TX_3	Tx V_p(r)	Tx V_q(p)	Tx V_r(q)
	RX_3	p2:=Rx V_p(V_q(p))	q2:=Rx V_q(V_r(q))	r2:=Rx V_r(V_p(r))
		q2:=Rx V_p(V_r(q))	r2:=Rx V_q(V_p(r))	p2:=Rx V_r(V_q(p))
V_PHASE_4	Vote	Vote(q1, q2)	Vote(r1, r2)	Vote(p1, p2)
		Vote(r1, r2)	Vote(p1, p2)	Vote(q1, q2)
		Vote(p1, p2)	Vote(q1, q2)	Vote(r1, r2)
IC_VOTE		Vote(p, q, r)	Vote(p, q, r)	Vote(p, q, r)

TABLE I
MESSAGE EXCHANGE BETWEEN NODES

Table I Legend:	
Action	**Description**
Tx p	it broadcasts the proprietary value p to the others nodes
q1:=Rx V_p(q)	it is the estimation made by a node (*p*) of a value (q) received from another node (*q*). The received value is assigned to a local variable (q1)
Tx V_p(q)	it broadcasts the value V_p(q) received at previous phase (relayed value)
r2:=Rx V_p(V_q(r))	it is the estimation made by a node (*p*) of a value (V_q(r)) received from another node (*q*). The received value is assigned to a local variable (r2)
vote(q1; q2)	it votes two previous estimated values

In the case of a system composed by four nodes the protocol will be modified adding a relay phase (C_PHASE_4).

B. The Generic model of Interactive Consistency protocol

The generic charts composing the model template for the proposed family are the following: TX, RX, vote.
In figure 4 there is the generic chart for a broadcasting transmitter: we can see that these basic components are extremely simple: actually the complexity of the overall system is obtained by replication and by the expansion of the generic terms in possibly complex expressions.

The chart has a generic name (STC_TX_ALL) a contextual symbol (TX_END) and two terms (T_RX_END, TX_VAR2BUS). The same structure is that of a generic receiver (see Fig. 5), also in this case there are: one generic name (STC_RX), a contextual symbol (RX_END) and two terms (T_TX_END, RX_BUS2VAR). All these generic symbols will be renamed or expanded.

The instantiation and expansion rules will be presented in next sections.

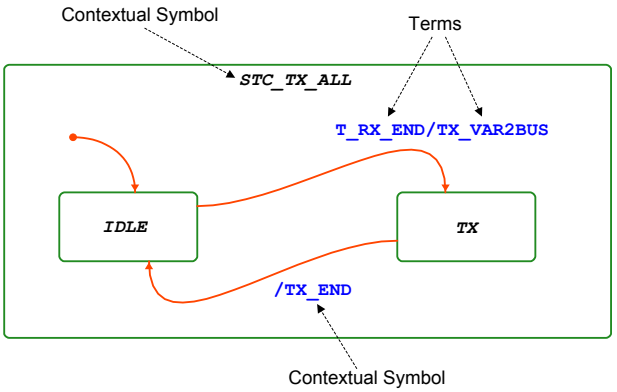

Fig. 4. Transmitter generic statechart

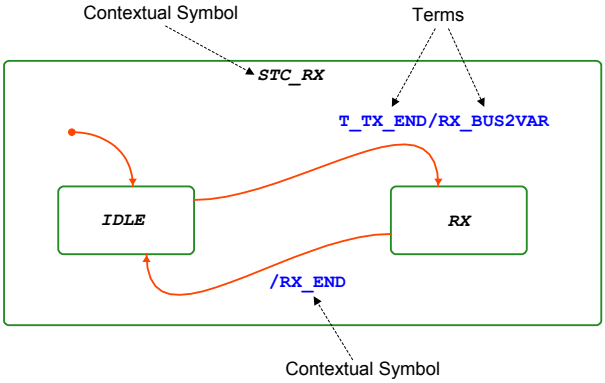

Fig. 5. Receiver generic statechart

C. The instantiation process applied to the case study

Referring to our case study, we need as many transmitters as the number of nodes and phases; it is evident that the behavior, derived by the states pattern, is the same for each transmitter, but they differ for conditions and actions, which depend on some parameters: the specific node instance, phase, number of nodes, and so on. Figure 4 shows an example of a generic statechart for the function of broadcast transmission from a node to the others. Figure 5 shows the corresponding generic receiver. Similarly we have defined a generic statechart for the voter component.

C.1 First step

Following the process described in section III, the first step is the replication of charts and contextual symbols renaming.

In Figures 4 and 5 we can identify four contextual symbols:

Generic contextual symbols:		
TX	STC_TX_ALL	generic statechart name
	TX_END	a variable
RX	STC_RX	generic statechart name
	RX_END	a variable

In Figures 6 and 7 two instances of statecharts are shown: one transmitter (node q, phase 2) and one receiver (link from node q to node p, phase 2).

While the instances are replicated contextual symbols are renamed by adding the proper suffix; at the end of first step in the example they become (see figures 6 and 7):

Instantiated contextual symbols:	
TX	STC_TX_ALL_Q_2
	TX_END_Q_2
RX	STC_RX_Q_P_2
	RX_END_Q_P_2

The added suffixes are one of the results of the following queries:

```
SELECT tx_name FROM tx
SELECT rx_name FROM rx
```

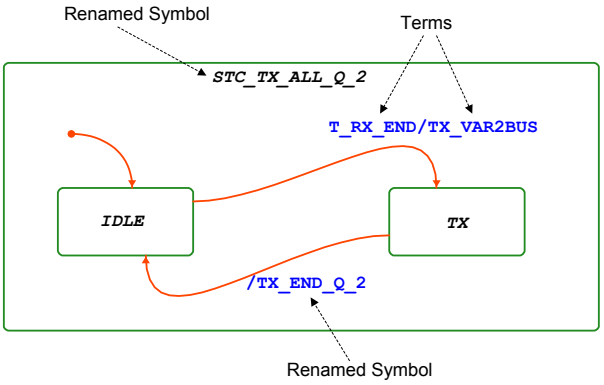

Fig. 6. The node q transmitter instantiated statecharts.

The database for the 3 nodes case study is given essentially by the information represented in Table 1, which is however shown in a more comprehensible style w.r.t. the actual data in the database.

C.2 Second step

The second step of the methodology consists in expanding terms, also in this step, it is implemented by queries. These queries are parameterized and their results change depending on which instance they are applied.

The generic terms are the following:

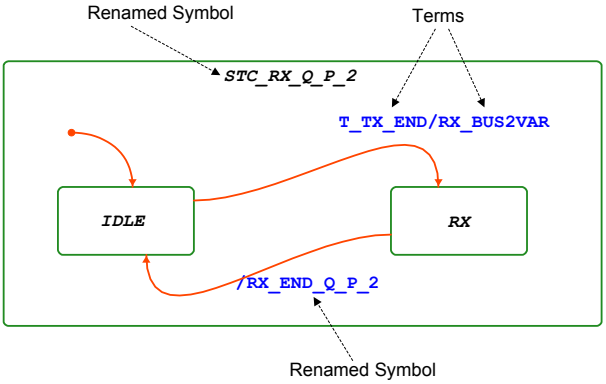

Fig. 7. The node q receiver instantiated statecharts.

Generic terms:		
TX	T_RX_END	events of termination of previous reception phase
	RX_BUS2VAR	assign values from links to local variables
RX	T_TX_END	event of termination of previous transmit phase
	TX_VAR2BUS	assign values from local variables to links

This expansion process has been driven by structured expansion rules and implemented by queries.

Here are shown two example of queries we have used to expand two terms.

$T\_TX\_END \leftarrow$

```
SELECT DISTINCT tx_terms.T_RX_END
FROM tx_terms WHERE
(((tx_terms.Node)='Q') AND
((tx_terms.PhaseTX)='2'));
```

$TX\_VAR2BUS \leftarrow$

```
SELECT DISTINCT tx_terms.TX_VAR2BUS
FROM tx_terms WHERE
(((tx_terms.Node)='Q') AND
((tx_terms.PhaseTX)='2'));
```

The resulting expansion is (see Figg. 8 and 9):

Generic terms:		Expanded terms:
TX	T_RX_END	RX_END_Q_R_1 and RX_END_Q_P_1
	RX_BUS2VAR	P_Q=R1; P_R=R1
RX	T_TX_END	TX_END_Q_2
	TX_VAR2BUS	R2=Q_P

To drive these expansions the following abstract rules

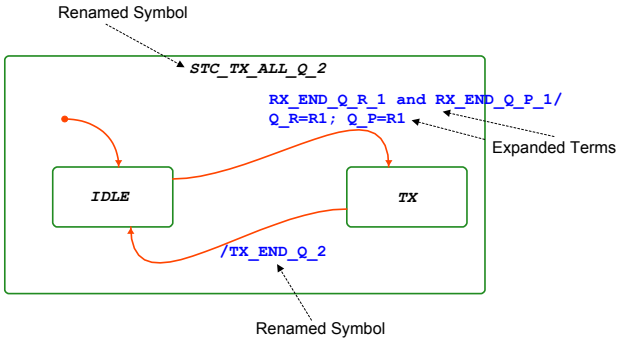

Fig. 8. The node *q* transmitter instantiated statecharts after expansion.

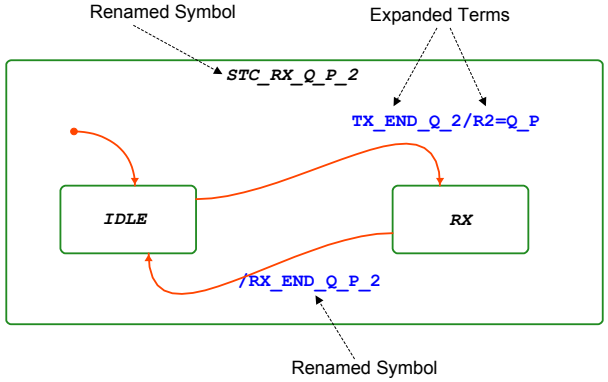

Fig. 9. The node *q* receiver instantiated statecharts after expansion.

have been used:[1]

`if` {*this is the first phase*} `then`
$\{T\_RX\_END \leftarrow START\_ALG\}$
`else` $\{T\_RX\_END \leftarrow \bigwedge_{ResQuery}[RX\_END\_p\_q\_i]\}$

where *p*, *q* are nodes and *i* is a phase, in particular RX_END_p_q_i are the variables signalling the termination of previous reception phases. For each transmitter in the first phase the term is expanded in a global variable named START_ALG which commands the start of the algorithm. Differently, for the following phases the term expansion substitutes it with more than one variables, these variables depending on the node and the phase.

$$RX\_VAR2BUS \leftarrow; [p\_q = Rj]$$

where *p*, *q* are nodes and *j* is a variable index, and the semicolon indicates the sequencing operator applied to all its arguments. In this case the sending of the Rj message over the p_q link is generated.

$$T\_TX\_END \leftarrow TX\_END\_p\_i$$

[1]Actually, the concrete expansion rules are queries to the knowledge coded in the database.

where *p* is a node and *i* is a phase. The term is substituted with a variable representing the end of the transmission phase.

$$TX\_BUS2VAR \leftarrow Rj = p\_q$$

where *p*, *q* are nodes and *j* is a variable index. In this case the reception of the messages from the p_q link is generated.

Parametrization is evident comparing figures 4 and 8, in fact in figure 4 the labels on the transitions have no references to particular objects of the topology. The association between this generic chart and the final instantiated chart has been done basing on the topology information which depends on the particular implemented application. In this case study the differences between applications of the same family are mainly related to the number of nodes.

Fig. 8 shows the statechart that has been instantiated for a transmitter (node *q*, phase 2), according to Table I. The chart related to one receiver is shown in detail as well (figure 9).

D. *The instantiated model of Interactive Consistency protocol*

The specification of the overall system, that is, the distributed Interactive Consistency protocol, is obtained combining the instantiated elements. In figure 10 is shown an architecture we have used to model the instantiated flat model into the Statemate tool, it uses a hierarchy of nested activity charts where leaves are the instantiated statecharts.

In the developed model with 3 nodes a set of 36 statecharts have been instantiated: 9 transmitter charts, 18 receiver charts and 9 voting charts.

In the case of a model with 4 nodes in each phase a receiver statechart has to be added. The four nodes model generated following the methodology will be formed by 80 statecharts: 16 transmitter charts, 48 receiver charts and 16 voting charts.

The obtained statechart specification of the Interactive Consistency algorithm has the same structure of the one presented in [3], where a process algebra was used instead, with the aim of performing a formal verification of the algorithm. In that case, the writing of the specification for both the three and four node cases were done by hand. Also in this case, verification tools working on statecharts (such as the Statemate model checker) can be used on the obtained models if this is desired. Depending on the aim of the model, using other tools available over statecharts, such as simulators or code generators, may be of interest: we were able for instance to simulate the obtained models on Statemate [9]. Given the ability of this simulator to interact with the user at run time, injection of faults can be carried out to verify its behavior under faults scenarios.

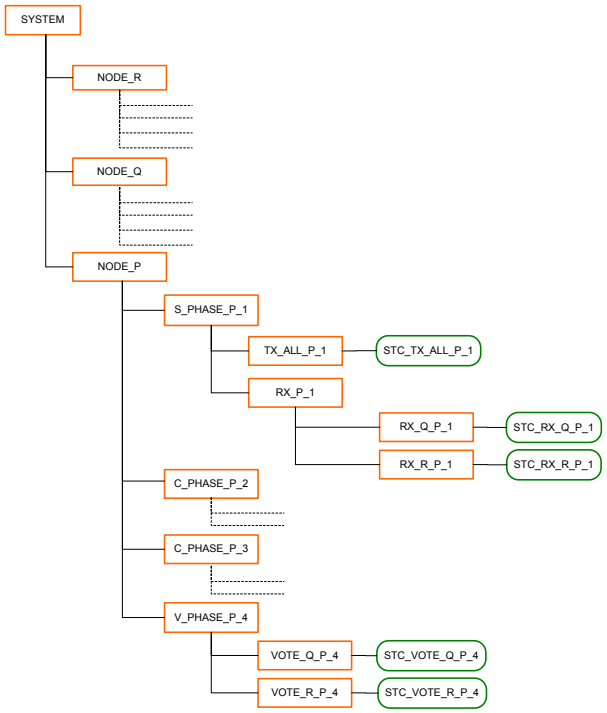

Fig. 10. Architecture hierarchy.

V. Conclusions

We have presented a methodology to build statechart models of topologically distributed systems by instantiating generic components; the method is able to replicate as many as needed statechart components and to expand the terms that drive their behavior to create complex interconnection patterns along the required topology. The method is particularly suitable for modeling a family of systems that can be obtained by replicating components of a few types: different configurations of the system may vary just for the number of components and for the topology of their interconnections. We have exemplified the approach over a classic generic distributed algorithm from the literature (Interactive Consistency), of which we have considered two different instantiations, built by reusing a few basic components.

The methodology has however been applied also to an industrial case study, related to a railway interlocking system, such as those identified in [1]. In this case we have produced a model constiuted by 267 statecharts, building on a set of basic statecharts (modeling the simplest devices of a railway interlocking) of 4 statecharts. This exercise has proved the scalability of the methodology to quite large designs, and has required the development of a prototype instantiator tool, capable of taking as input some generic statecharts and a description of the application topology, and to produce a statechart model of the system targeted to the considered topology.

The availability of an instantiator tool is thought to produce a positive effect of acceptability in an industrial context, due to minimization of manual work. On the basis of this experience, we believe that the proposed approach may find many useful industrial applications, in all the cases where a model of some large statically configured (distributed) systems is needed, and especially when a family of products, each depending on a specific (logical or physical) topology of components, has to be defined, with the added value of the availability of formal specification and verification tools.

Ongoing work on the tool is aimed at generalizing to different application domains, and at achieving a full integration with commercial specification and verification tools (such as Statemate, Stateflow, UML tools, ...) for the various dialects of statecharts, which provide simulation, verification, and code generation capabilities.

One of the aim of producing a model of a large system is to be able to simulate it and to check over the model the satisfaction of some required properties, such as safety properties, before actually producing the system. Indeed, the large dimensions that can be achieved for a model produced following the proposed methodology challenge the current capability of verification tools. Besides experimenting with the actual limits of the verification tools, we plan to investigate the problem of properties preserved in the instantiation process: we would like to establish which properties, satisfied by the generic statecharts, are (maybe once properly transformed) preserved in the instantiated model. This knowledge would allow to address the above mentioned scalability problems of verification.

References

[1] M. Banci, A. Fantechi, *Geographical vs. Functional Modelling by Statecharts of Interlocking Systems*. FMICS Ninth Workshop on Formal Methods for Industrial Critical Systems, Linz, September 20-21, 2004. Electronic Notes in Computer Science (Elsevier).

[2] C. Bernardeschi, A. Fantechi, S. Gnesi, *"Formal verification", Chapter 10 of D. Powell ed., "A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems"*, Kluwer Academic Publishers, Boston, ISBN 0-7923-7295-6, January 2001

[3] C. Bernardeschi, A. Fantechi, S. Gnesi, *"Formal validation of the GUARDS Inter-consistency mechanism"*, SAFECOMP'99, Tolosa, September 1999, Lecture Notes in Computer Science, vol. 1698.

[4] P. C. Clements, L. Northrop. *Software Product Lines: Practices and Patterns.* SEI Series in Software Engineering. Addison-Wesley, August 2001

[5] D. Harel, M. Politi, *Modelling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998

[6] L. Lamport, R. Shostak, M. Pease. *The byzantine generals problem.* ACM Transactions on Programming Languages and Systems, 1982; 4(3): 382-401.

[7] F. van der Linden. *Software Product Families in Europe: The Esaps and Café Projects.* IEEE Software, 19(4):41–49, July-August 2002.

[8] *The Mathworks: Stateflow and Stateflow Coder, Users Guide.* Release 13sp1 edn. (2003)

[9] *Statemate Magnum Simulation Reference Manual.* I-Logix Inc. Burlington, MA USA, 2003.

[10] Object Management Group, *Unified Modelling Language Specification, Version 1.5, 1999*
http://www.omg.org/technology/documents/formal/uml.htm