From Requirements to Statecharts via Design Refinement

João Pimentel, Jaelson
Castro
Universidade Federal de
Pernambuco - UFPE
Recife, Brazil
jhcp@cin.ufpe.br

John Mylopoulos,
Konstantinos
Angelopoulos
University of Trento - UniTn
Trento, Italy
jm@disi.unitn.it

Vítor E. Silva Souza
Universidade Federal do
Espirito Santos - UFES
Vitória, Brazil
vitorsouza@inf.ufes.br

ABSTRACT

The derivation of statecharts from requirements has been addressed from many perspectives. All of them assume that the derivation process is a linear series of refinements resulting in a single statechart, thereby missing the opportunity to explore alternatives in the design space. We propose a multi-dimensional approach that exploits inherent variability of the design space, where alternative refinements are considered for the same intermediate problem, resulting in multiple solutions (statecharts) from a single initial problem (requirements). In order to accomplish this, we propose an extended form of goal model where architects can incrementally refine the original requirements by considering behavioral alternatives leading to design solutions. The proposed refinement process is illustrated through an example from the literature. Experimentation with randomly generated models suggests that the proposal is scalable.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques

1. INTRODUCTION

Requirements engineering and architectural design, while addressing the system specification at different abstraction levels, are intertwined activities. The former focuses on the problem at hand, whereas the latter provides solutions for that problem [14]. One of the challenges in Software Engineering (SE) is to determine whether a system specification is actually a solution to the given requirements. We propose to tackle this challenge by systematically deriving the latter from the former. In earlier work we proposed a process for deriving architectural models from requirements [4][15] . However, these proposals handle only the structural view of the architectural design, lacking any behavioral specification of the system-to-be. The behavioral view accounts for the

order of execution of tasks that fulfills requirements, including concurrency and time dependencies for the system-to-be [2]. Besides, it allows further refinement/design of the system and supports reasoning to detect potential deadlocks, termination and efficient use of scarce resources.

In this work we propose a systematic process for deriving behavioral models — expressed as statecharts [8] — from goal models through a series of refinements expressed within an extended goal model that constitutes an intermediary model between requirements and architecture, similar to [7]. By exploiting the use of views, we reduce the burden of maintaining an additional model. A key contribution of this work is that, through the definition of alternative refinements for the same design element, this model captures the inherent variability of the design space, where different solutions (statecharts) for a given problem can be devised. Moreover, the ability to automatically generate statecharts from partially refined models facilitates incremental design and the intertwined refinement of requirements and architecture [14]. To evaluate this process, a case study was developed based on an Automated Teller Machine (ATM), a system extensively described in the software engineering literature, which is also used as running example in this paper. This case study was performed using a prototype tool developed to support this process. Furthermore, in order to analyze the applicability of deriving statecharts for large models, we performed a scalability assessment of the automatic generation of statecharts from design goal models, presented in Section 4.

2. BASELINE

The following sub-sections provide a brief overview of the baseline of this proposal.

2.1 Goal Modeling

Requirements can be represented as goals, tasks, domain assumptions (DAs) and quality constraints (QCs). These elements are supported by many requirements modeling approaches . The space of alternatives for goal satisfaction is represented by Boolean AND/OR refinements with obvious semantics. Tasks, on the other hand, are directly mapped to functionality in the running system and are satisfied if executed successfully. However, QCs can define additional restrictions for the satisfaction of goals and for the execution of tasks. Finally, DAs are satisfied if they hold (affirmed) while the system is pursuing their parent goal.

Fig. 1 shows a goal model with the requirements for our running example, an Automated Teller Machine (ATM) sys-
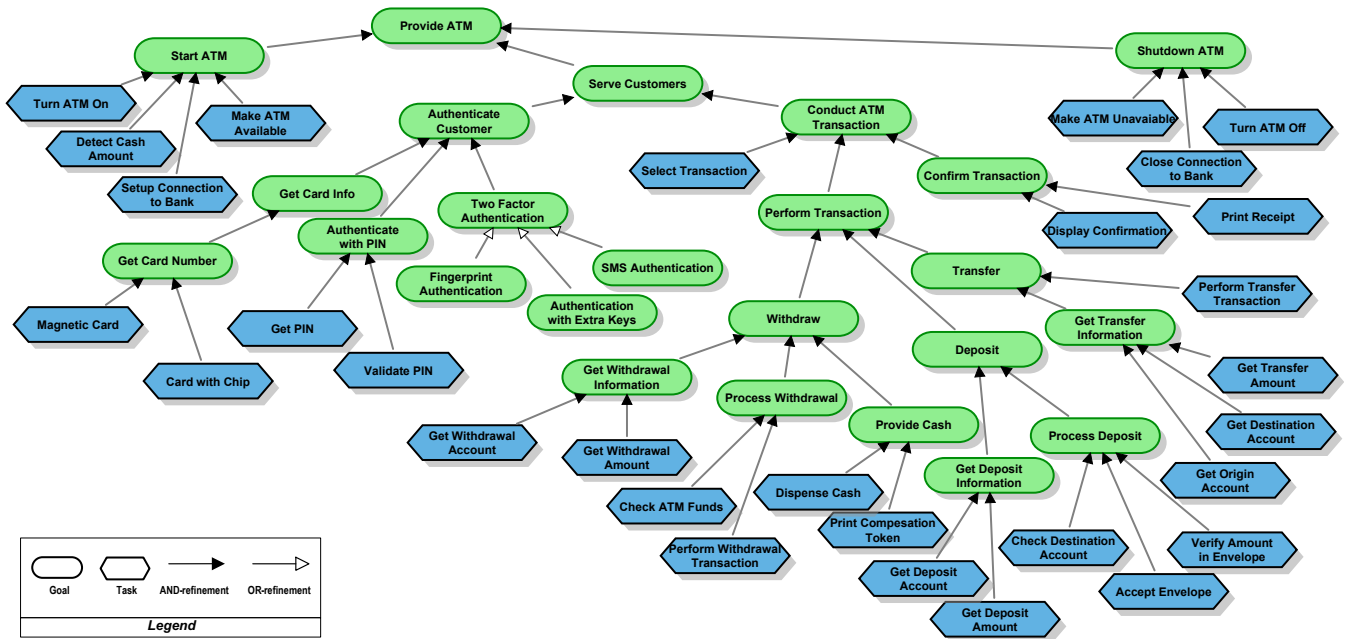
Figure 1: ATM Requirements Model. QCs omitted for the sake of space

tem. This model was based on a model from [19]. In order to provide its services, the ATM needs to be turned on and off, as well as to *Serve Customers*. This goal is refined into *Authenticate Customers* and *Conducting Transactions*. In order to conduct transactions, it is required to *Select Transaction*, *Perform Transaction* and *Confirm Transaction*. The transactions available to be performed are *Withdraw*, *Deposit* and *Transfer*.

2.2 Flow Expressions

Flow expressions [16][5] describe the flow of system behavior in terms of extended regular expressions. In our use of such expressions we adopted different symbols in order to facilitate their writing (e.g., | in place of ∪ for expressing alternative behaviors). Each atomic symbol represents an element of the flow, which in our case is an element from the goal model. For example, if $g1$ is a goal, the atomic expression `g1` represents the state where the goal $g1$ has been fulfilled. Flow expressions can be composed in terms of regular expression operators, such as concatenation (`g1 g2`), meaning "first satisfy $g1$ then $g2$" (sequential flow), or `g1*`, meaning that $g1$ is to be satisfied zero or more times.

Flow expressions separated with a vertical bar | symbol represent alternative flows. The question mark ? is used to represent the optionality of the flow to its left, i.e., that flow may be executed zero or one times. The star symbol * indicates that the flow to its left may be executed zero, one or more times, while the plus symbol + indicates that that flow may be executed one or more times. The shuffle operator (here expressed as the sharp symbol #) indicates that two flows are to be carried out concurrently, in the sense that their states can be interleaved. Considering the letters from A to H as atomic flows, the flow expression

$$(A \; B \; (C \mid D) \; E \; F* \; G) \; \# \; (H*)$$

indicates that state A is followed by state B. After B, the possible states are C or (exclusively) D, followed by E. After

E, F may be reached any number of times. State G may be entered after E or after F. Concurrently to all that, the state H may occur any number of times.

2.3 Statecharts

Statecharts [8] are a visual formalism that can be used to specify architectural behavior [2]. The main elements of this modeling language are states that our system-to-be can be in, and transitions that represent possible state changes. A transition has an associated event that triggers the transition, and a condition that must be true for the transition to occur. Thus, by specifying a graph of states connected by transitions it is possible to specify how the system-to-be reacts to different events, depending on its current state.

Unlike their finite state machine cousins, statecharts allow to structure states in a hierarchy of super- and sub-states. In an XOR state, if the state is active, so is one and only one of its sub-states; in an AND state, if a state is active so are all of its sub-states. The AND state (concurrent) is represented with a dashed line separating its sub-states.

3. FROM GOALS TO STATECHARTS

In order to move from requirements to statecharts, we advocate the use of extended goal models which capture both requirements and design concerns. This design goal model extends the goal model presented in Section 2.1 with (i) design tasks and design quality constraints, (ii) assignments, and (iii) behavioral refinements. The complete process comprises five steps: *Identify design tasks and constraints; Assign tasks; Define basic flows; Generate base statechart*; and *Specify transitions*. While these steps can be followed sequentially, waterfall-like, in realistic settings it is expected that the architect will go back and forth, by introducing additional refinements to already refined elements. Moreover, during the derivation process there may be a need to revisit the original requirements in order to introduce new refine-

ments because those currently pursued are not working out. In fact, in accordance with the Twin Peaks directive [14], the approach supports the generation of statecharts from partial design goal models, i.e., from models that have not been fully refined all the way down to its leaf elements. For instance, if one defines the behavioral refinement only of the root goal, the result would be a very high-level statechart. As further refinements are defined and new elements are added to the model, the resulting statechart will get more complete. Thus, it is not necessary to complete one step in order to proceed to the next one.

The steps of the process are presented with further details in the following sub-sections.

3.1   Identify Design Tasks and Constraints

Requirements expressed as goal models describe the problem space for the system-to-be, capturing concerns from different stakeholders such as customers, users and domain experts. During design, i.e., as we move towards the solution space, the different elements of the requirements model need to be further refined, reflecting design decisions that have been made. These decisions can be classified in three categories: existence decisions, property decisions, and executive decisions [9]. Although some of these impact the system as whole (e.g., the selection of an architectural pattern), some others can be traced to specific elements of the requirements model.

In order to capture design decisions that refine how a certain goal can be achieved, how a certain task can be performed, and how a given quality constraint can be satisfied, we propose the inclusion of *design tasks* and *design quality constraints* in the goal model. As argued in [6], there is a large similarity between architecturally significant requirements and design decisions. Nonetheless, we opt to differentiate design elements from their requirements cousins in order to make it clear, among other things, (i) who (stakeholders or designers) is responsible for making decisions with respect to those elements, and (ii) in which stage of the project they appear. This differentiation is done by using dashed borders for the design elements (e.g., *design tasks* in Fig. 2).

Design tasks and *design quality constraints* are included in the goal model through AND/OR refinements. By including these elements in the goal model, rather than using a separate design decisions model, we can take advantage of existing goal reasoning infrastructure when designing systems with specific needs such as self-adaptation [18] and context-awareness [10]. In Fig. 2 we present excerpts of the refinements for the ATM system. The *Get PIN* task was refined with three different alternatives for getting the PIN code: through a regular keypad, through a 2-key keypad and through a touchscreen keypad. For the *Detect Cash Amount* task, two alternatives were identified: *Use Cash Sensor* and *Use Operator Entry*.

3.2   Assign Tasks

The goal model presents the tasks that need to be performed by the system-to-be, but does not prescribe who is responsible for performing them: human actors, organizations, software systems, etc. This step of the process consists of defining this responsibility. A task may be assigned to one or more actors, with the meaning that it may be performed by any one of them. If, instead, part of the task
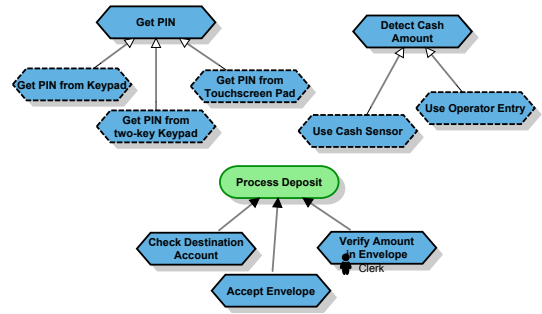

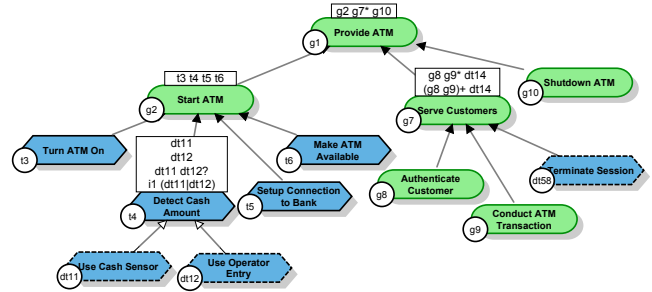**Figure 2: Excerpts of the ATM Design Model, with Design Tasks and Assignment**

Figure 3: Behavior refinements for an excerpt of the ATM Design Model

will be performed by an actor and another part by another actor, the original task can be further refined with sub-tasks assigned to different actors. Fig. 2 shows the assignment of the *Verify Amount in Envelope* task to a bank clerk.

3.3   Define Basic Flows

In this step we define the order of goal fulfillment and task execution, through the use of flow expressions. This is done by, for each element that will be refined, defining a flow expression which describes the behavior of its children elements. By constraining this decision-making to descendants (children) of a node, instead of considering the system as a whole at once, it is easier to define the flows.

Before defining the flow it is important to reconsider the AND refinements present in the original model. At the requirements level an AND refinement means that the system must support functionality for fulfilling all the children elements of that refinement. At runtime, though, this does not mean that all of these children need to be executed every time in order to achieve the parent goal. This is the case, for instance, on the refinement for the *Perform Transaction* goal of the ATM system. This kind of system must provide both deposit, withdraw and transfer capabilities, which can be expressed as an AND-refinement in the requirements model (Fig. 1). At runtime, however, the user may select only to deposit or only to withdraw (among other options). Additionally, the user may select to perform a task repeatedly, which calls for the need to express multiplicity of execution.

These cases are handled by refactoring the goal model, moving the element to the appropriate level in the tree.

Fig. 3 shows the flow expressions for an excerpt of the ATM system. The flow expression from the root goal *Provide ATM (g1)* indicates that its behavior is defined as start-

ing the ATM *(g2)*, then serving customers zero, one or more times *(g7)*, and finally shutting down the ATM *(g10)*. Similarly, the behavior of *Start ATM (g2)* is defined by the sequential execution of its children tasks (from *t3* to *t6*).

In the same way that OR refinements of goals can express alternative ways for achieving a goal, it is possible to define alternative behavioral refinements. This is the case of the *Serve Customers* goal *(g7)*, with two options, and of *Detect Cash Amount (t4)*, with four alternatives, totaling eight different statecharts that can be generated from that excerpt. The requirements model dictates that to be able to *Serve Customers* the system must provide functionalities to *Authenticate Customer* and *Conduct ATM Transaction*. There are different behaviors for this goal, as shown in Fig. 3. The first option is to authenticate customers *(g8)* only once, and then conduct ATM transactions *(g9)* any number of times. The second option is to authenticate customers *(g8)* before conducting each transaction *(g9)*.

This is also the case for *Detect Cash Amount (t4)*, which has four alternative behavioral refinements. The first option is to use only a cash sensor *(dt11)*, while the second option is to use only the entry of an operator *(dt12)*. The third option is to use the cash sensor and then use the operator entry only if it is necessary (e.g., if there is a malfunction on the cash sensor). The last alternative includes the use of an intermediate state where the operator can select whether to use the cash sensor or to entry the value manually. The use of intermediate states is a common practice when creating statecharts, to define points where the system is waiting for some input, e.g., waiting for a selection by the user. These states are included in the flow expressions using the '*i*' prefix. As future work, we plan to identify the most common types of intermediate states and create pre-defined constructs for them.

Considering that there are different ways for the system to perform a set of tasks, determining the behavioral refinement (through flow expressions) is not a matter of direct translation, but rather constitutes an important design decision – it is, thus, influenced by non-functional requirements, reuse of components, previous decisions (for instance, regarding the architectural style for the system structure), among other factors.

It is important to note that it is possible to generate partial statechart models from partially refined models, as described in the following subsection. In other words, it is not necessary to define the behavioral refinement of the entire model in order to visualize the resulting statechart, which is consonant with the Twin Peaks notion of iteratively and incrementally refining requirements and architecture [14].

3.4 Generate base statechart

Considering the behavioral refinements defined in the previous step, it is now possible to generate a base statechart model, which presents a comprehensive view of the system behavior. In order to support the generation of the statechart, we define a set of derivation patterns related to the different flows that may be expressed — sequential, alternative and concurrent — as well as to their optionality and multiplicity. These patterns, grounded on the graphical representation from [16], are depicted in Fig. 4.

Since it is possible to define alternative flow expressions in the model, the architect first needs to select the flows that will be used for the derivation. The statechart resulting
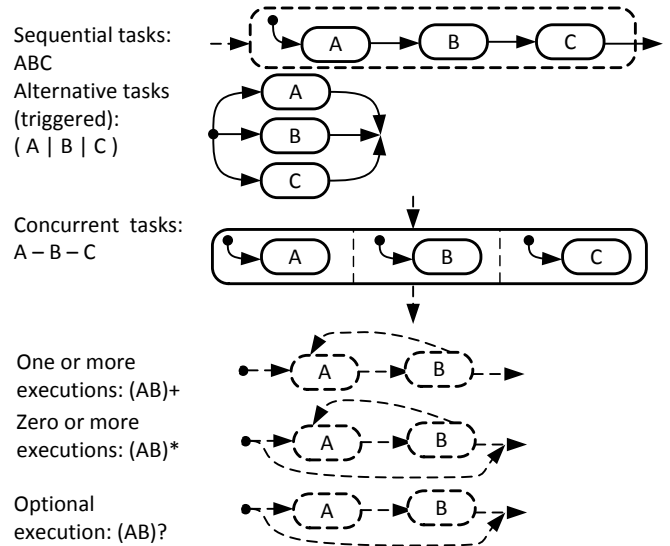

Figure 4: Visual representation of patterns for deriving statecharts from flow expressions

from applying the derivation patterns in the partial refinement presented in Fig. 3 is presented in Fig. 5A. For this statechart, we selected the third alternative refinement of *Detect Cash Amount* — *dt11 dt12?* — and the first alternative of *Serve Customers* — *g8 g9* dt14*. The goals which were not refined are highlighted in the statechart. Fig. 5B shows another version of the *Serve Customers* state, which represents its second alternative behavioral refinement — *(g8 g9)+ dt14*.

3.5 Specify transitions

So far we have defined the basic flow of the system, making it possible to check at runtime if the trace of tasks execution is valid. However, in order to define the system behavior in the face of optionality, multiplicity and alternatives, we need to know when a particular task should be triggered. Thus, in this step it is defined which events trigger a particular alternative, the execution of an optional task, as well as whether a flow should be repeated or not. It is also possible to define a combination of alternative events — for instance, a given task will be executed at 30 minutes intervals or upon user request.

Besides events, it is often useful to associate conditions to transitions, also adding transitions for cases where a condition is not satisfied. Defining such conditions is a task for the designers. Nonetheless, some elements of the requirements model can guide the definition of these conditions: domain assumptions, quality constraints, and other tasks (that constitute pre-conditions).

4. VALIDATION

In order to evaluate the derivation of statecharts we developed a prototype tool and used it to derive statecharts for the ATM system. When starting the derivation, the tool allows us to select between alternative behavioral refinements previously included in the models. Once selected, it traverses the design goal model generating a combined flow expression. For instance, considering that (g2 g3) is the
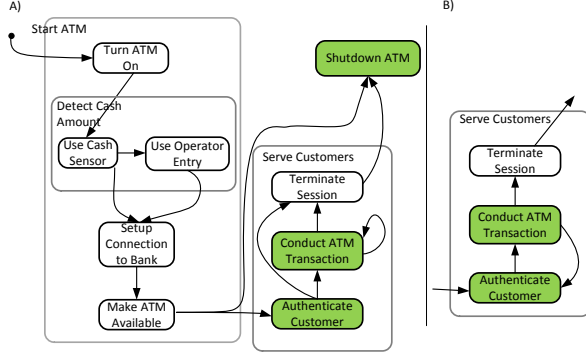
Figure 5: Statechart for a partial behavioral refinement of the ATM system

```
XOR-states: g1(g2,g7,g10), g2(t3,t4,t5,t6), t4(dt11,
    dt12), g7(g8,g9,dt58), g8(g14,g15,g53), g14(g16),
    g16(t17,t18), g15(t19,t20), g53(g54,g55,g56), g9
    (t24,g25,g26), g25(g29,g30,g31), g29(g37,g38,g39)
    , g37(t40,t41), g38(t42,t43), g39(t44,t45), g30(
    g46,g47), g46(t48,t49), g47(t50,t51), g31(g32,t33
    ), g32(t34,t35,t36), g26(t27,t28), g10(t21,t22,
    t23)
Atomic states: t3, dt11, t5, t6, t17, t18, t19, t20,
    g54, g55, g56, t24, t40, t41, t43, t42, t45, t44,
    t48, t49, t50, t51, t34, t35, t36, t33, t27, t28
    , dt58, t21, t22, t23
Transitions: ~->t3, t3->dt11, dt11->t5, t5->t6, t19->
    t20, t17->t19, t18->t19, t20->g54, t20->g55, t20
    ->g56, t40->t41, t43->t42, t41->t43, t45->t44,
    t42->t45, t42->t44, t48->t49, t50->t51, t49->t50,
    t34->t35, t35->t36, t36->t33, t24->t40, t24->t48
    , t24->t34, t27->t28, t42->t27, t45->t27, t44->
    t27, t51->t27, t33->t27, t27->t24, t28->t24, g54
    ->t24, g55->t24, g56->t24, g54->dt58, g55->dt58,
    g56->dt58, t27->dt58, t28->dt58, dt58->t17, dt58
    ->t18, t6->t17, t6->t18, t21->t22, t22->t23, t6->
    t21, dt58->t21
Concurrent states: none
```

Figure 6: Output of the statechart derivation for the ATM system

selected behavioral refinement for a goal *g1*, and (t4+ t5) and (t6|t7) are the selected behavioral refinements, respectively, for *g2* and *g3*, then the combined expression flow will read t4+ t5 (t6|t7). While traversing the model, the tool also identifies the XOR-states and its sub-states resulting from the combination, which in this example are *g1(g2,g3), g2(t4,t5)* and *g3(t6,t7)* — i.e., g1 is an XOR-state containing g2 and g3, g2 is a XOR-state containing t4 and t5, and g3 is a XOR-state containing t6 and t7.

The combined flow expression is then used by the tool to identify the states, transitions and AND-states (concurrent states). The output for the above expression, along with the XOR-states, is presented in Fig. 6, where a tilde (~) represents the initial state of the system. From the output, our statechart is defined as the tuple $\langle S, T, R \rangle$, where S is the union of the atomic states, the XOR states and the AND (concurrent) states; T is the set of transitions; and R is the union of the set of XOR state refinements and the set of AND state refinements.

Besides applying the process to the ATM system, we conducted an experiment to study the scalability of statechart generation algorithm. The experiment was conducted on a computer with a 64 bits Pentium Dual CPU T4300 processor with 2.1 GHz and 3 Gb of memory.

Table 1: Average time for deriving statecharts from flow expressions.

Size	Average time (ms)	$\sigma(\%)$
100	63.40	16.12
300	77.16	13.62
500	86.04	17.82
700	92.86	13.36
900	98.27	13.42

The inputs of this simulation are five flow expressions with different number of elements (100, 300, 500, 700, and 900). The first flow expression, with 100 elements, was randomly generated and then composed to create the larger expressions. These expressions use all possible operators: sequential, alternative, concurrent, optional, zero or more times, and one or more times. The derivation of each expression was executed 1000 times, cycling between the expressions, in order to reduce interference from the operating system. Table 1 presents the results of the experiment, with average execution times and deviation. These results indicate that the derivation algorithm is definitely scalable.

To complement this evaluation we plan to perform analysis with large industrial systems, as well as to assess the human effort required by this design process.

5. RELATED WORK

There are other approaches that use goal models as a basis for the definition of system behavior. In particular, the work by Yu et al. [20] is similar in spirit to ours. However, their statechart structure is derived directly from the structure of the goal model, thus not supporting the definition of inter-relation between elements that are in different sub-trees on the goal model. Both approaches are able to translate the variability present in a goal model to variability of a derived statechart. However, we support the definition of additional variability by supporting the design of alternative design tasks and alternative behavior refinements.

Liaskos et al. [13] support the definition of systems where the order of task execution is constrained at design time through Linear Temporal Logic (LTL) expressions. The resulting models are intended to support customization of system behavior, whereas here we focus on system behavior design. Letier et al. [11] start with KAOS models and derive Labeled Transition Systems (LTS), which have a formalization akin to statecharts. The goal models in KAOS present a temporal formalization in LTL, thus the derivation of behavioral models is mostly (but not only) a translation of formalism, with little human input.

There are some other relevant approaches for architectural derivation from requirements. The SIRA approach [3] guides the derivation of architectures from requirements based on *i** models. In Silva et al. [17] a set of mapping rules is proposed between Aspectual Oriented V-graph (AOV-graph) and AspectualACME. The CBSP approach [7] supports the derivation from requirements to architecture. The STREAM-A approach [15] presents a systematic approach for deriving ACME models from *i** models in the context of adaptive systems. All these approaches differ from ours in the sense that they do not support the derivation of behavioral models of the architecture, focusing on its structure.

6. CONCLUSION

We have presented a systematic process for deriving architectural behavioral models — namely, statecharts — from requirements models, supporting the Twin Peaks model [14] of software design. Through a series of incremental refinements the architect can move towards architecture, by (i) adding design elements (tasks and quality constraints), (ii) adding behavioral refinements (in the form of flow expressions), and (iii) generating statechart models from (possibly incomplete) models. Acknowledging the inherent variability of the design process, where different solutions for a single problem can be devised, the design goal model supports the documentation of alternative design elements and alternative behavior refinements, which can lead to the generation of multiple (alternative) statecharts. Since the resulting models are statecharts without any extension, they are amenable to validation, simulation and code generation using existing tools. Moreover, the properties of these models can be checked for consistency using one of the several formalizations of statecharts (for instance, [12]).

A key element of our proposal is the integration of requirements and design elements in a single model. The use of different views in the supporting tool allows to seamlessly navigate between requirements and design elements. We plan to further advance this approach to support the development of self-adaptive systems, adopting goal model extensions such as awareness requirements and control variables [18]. This will allow us to support both requirements-based and architectural-based adaptation [1].

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] K. Angelopoulos, V. E. S. Souza, and J. Pimentel. Requirements and Architectural Approaches to Adaptive Software Systems: A Comparative Study. In *Proc. of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (to appear)*, 2013.

[2] F. Bachmann, L. Bass, P. Clements, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. Documenting Software Architecture : Documenting Behavior. Technical Report January, 2002.

[3] L. R. D. Bastos and J. Castro. From requirements to multi-agent architecture using organisational concepts. *SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[4] J. Castro, M. Lucena, C. Silva, F. Alencar, E. Santos, and J. Pimentel. Changing attitudes towards the generation of architectural models. *Journal of Systems and Software*, 85(3):463–479, Mar. 2012.

[5] F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos. Runtime Goal Models. In *Proceedings of the 7th IEEE International Conference on Research Challenges in Information Science (RCIS 2013)*, 2013. Invited paper.

[6] R. C. de Boer and H. van Vliet. On the similarity between requirements and architecture. *Journal of Systems and Software*, 82(3):544–550, Mar. 2009.

[7] P. Grünbacher, A. Egyed, A. Way, W. Place, M. D. Rey, and N. Medvidovic. Reconciling software requirements and architectures: the CBSP approach. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 202–211. IEEE Comput. Soc, 2001.

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

[9] P. Kruchten. An Ontology of Architectural Design Decisions in Software-Intensive Systems. *2nd Groningen Workshop Software Variability*, pages 54–61, 2004.

[10] A. Lapouchnian and J. Mylopoulos. Modeling domain variability in requirements engineering with contexts. *Conceptual Modeling - ER 2009 - LNCS*, 5829/2009:115–130, 2009.

[11] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engineering*, 15(2):175–206, May 2008.

[12] F. Levi. *Verification of Temporal and Real-Time Properties of Statecharts*. PhD thesis, University of Pisa, 1997.

[13] S. Liaskos, S. M. Khan, M. Litoiu, M. D. Jungblut, V. Rogozhkin, and J. Mylopoulos. Behavioral adaptation of information systems through goal models. *Information Systems*, 37(8):767–783, Dec. 2012.

[14] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–119, Mar. 2001.

[15] J. Pimentel, M. Lucena, J. Castro, C. Silva, E. Santos, and F. Alencar. Deriving software architectural models from requirements models for adaptive systems: the STREAM-A approach. *Requirements Engineering*, 17(4):259–281, June 2012.

[16] A. C. Shaw. Software Descriptions with Flow Expressions. *IEEE Transactions on Software Engineering*, SE-4(3):242–254, May 1978.

[17] L. F. Silva, T. V. Batista, A. Garcia, A. L. Medeiros, and L. Minora. On the symbiosis of aspect-oriented requirements and architectural descriptions. *Early Aspects: Current Challenges and Future Directions - LNCS*, 4765/2007:75–93, 2007.

[18] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness Requirements. In R. Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 133–161. Springer, 2013.

[19] G. Tallabaci and V. E. S. Souza. Engineering Adaptation with Zanshin: an Experience Report. In *Proc. of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (to appear)*, 2013.

[20] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C. S. P. Leite. From Goals to High-Variability Software Design. In *Foundations of Intelligent Systems*, volume 4994/2008, pages 1–16, 2008.