Automated Encapsulation of UML Activities for Incremental Development and Verification

Frank Alexander Kraemer and Peter Herrmann

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway
{kraemer,herrmann}@item.ntnu.no

Abstract. With their revision in the UML 2.x standard, activities have been extended with streaming parameters. This facilitates a reuse-oriented specification style, in which dedicated functions can be contributed by self-contained activities as building blocks: Using streaming parameters, activities can be composed together in a quite powerful manner, since streaming parameters may also pass information while activities are executing. However, to compose them correctly, we must know in which sequence an activity may emit or accept these streaming parameters. Therefore, we propose special UML state machines that specify the externally visible behavior of activities. Further, we develop an algorithm to construct these state machines automatically for an activity based on model checking. Using these behavioral contracts, activities can then be composed without looking at their internal details. Moreover, the contracts can be used during system verification to reduce the complexity of the analysis.

Keywords: System Composition, UML Activities, UML State Machines, UML Streaming Parameters, Model Reuse, Verification.

1 Introduction

UML activities can be used on several levels of decomposition for the specification of systems. On a high level, activities may cover coarse business processes and provide the big picture of a system's behavior. Activities are also equipped with the necessary concepts to express fine-grained logic on a more detailed level, close to an implementation in a programming language. These different levels of abstraction are not in conflict with each other, and can all be part of a consistent specification: By using *call behavior actions*, an activity may refer to subordinate activities, so that a complete system specification may be decomposed on numerous levels, from the high level focusing on the overall behavior, towards such a degree of detail that code can be generated from them.

When referred to via call behavior actions, activities may pass data and control flow between each other using input and output parameter nodes. With version 2.0 of the UML standard [1], activity parameter nodes were extended with the concept of *streaming* parameters. While non-streaming parameters may

A. Schürr and B. Selic (Eds.): MODELS 2009, LNCS 5795, pp. 571–585, 2009.
© Springer-Verlag Berlin Heidelberg 2009

only accept tokens at the start or emit tokens at the termination of an activity, streaming parameters may pass tokens throughout the execution of an activity, in any order and frequency. This enables more elaborate dependencies between activities, so that related functionality can still be encapsulated within one activity, but a detailed synchronization between those activities is enabled by using streaming parameter nodes. This is a form of interleaving composition, and from the experience gained from our case studies introduced later we have seen that enabling this composition fosters the reuse of activities in the form of building blocks.

To effectively exploit the potential of interleaving compositions enabled by streaming parameter nodes, however, we need a description of the external behavior of an activity relevant for an enclosing context. This is a kind of interface, hiding the internal details of an activity. For this purpose, we complement activities with so-called *External State Machines* (ESMs) which are a variant of UML state machines. An ESM describes the order in which tokens can pass the various parameter nodes of an activity. This order has to be obeyed to guarantee a correct interplay between an activity and its environment. The concise notion of the external behavior of activities by ESM offers a number of advantages for the incremental development and verification of system specifications:

- Developers reusing an activity do not have to consider its internal details, but may rely on the description given by its ESM.
- The formal interface description described by an ESM can be used to verify that the activity is correctly embedded in a surrounding specification.
- ESMs support the incremental development of systems. In a bottom-up style, activities can be encapsulated by ESMs, facilitating their composition to more comprehensive models since details are hidden. In a top-down style, ESMs can be used to first sketch the external behavior of an activity, which can be subsequently implemented separately from a global model, just by considering its ESM.
- ESMs can be used to guard changes in models. The internals of an activity can be modified without affecting models referring to it if it still complies with its ESM, which can be verified automatically by tools.
- ESMs enable incremental verification. During a formal analysis of a system specification based on model checking, activities can be analyzed separately. This reduces the state space needed during the analysis significantly. Moreover, once an activity is verified, this verification does not have to be repeated when the activity is reused. The surrounding context only has to comply with its ESM.

As one realization of a model-driven development process using UML activities and ESMs, we have proposed the engineering method SPACE [2,3], depicted in Fig. 1. Systems are specified as hierarchies of activities encapsulated by ESMs. Those activities useful in several applications are stored together with their ESMs as self-contained building blocks in libraries for different domains. Currently, we have libraries for embedded sensor systems [4], trust management [5], and web service-based telecom services [6].

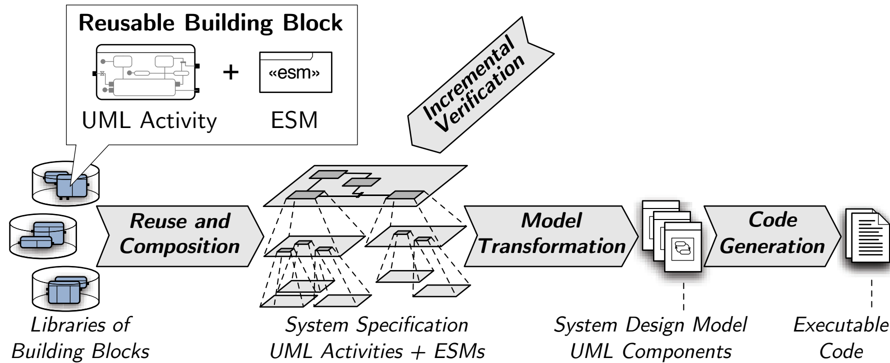

Fig. 1. Model-driven development method based on UML activities and ESMs

The method aims at a high degree of automation. With the tools described in [7,8], activities and their compositions can be checked automatically for numerous properties that should hold for any applications. This analysis is performed incrementally, i.e., on each activity separately and utilizes the reduction of state space as provided by the ESMs. To automatically implement the systems specified by activities, we developed and implemented a transformation algorithm that synthesizes UML state machines and composite structures [9,10]. From these state machines, we generate code for different execution platforms, for example for Java in different editions [11,12].

In this article we focus on the encapsulation of activities by ESMs, how this process can be automated, and present the impact on the development and verification from our case studies. In the following two sections, UML activities and ESMs are introduced. Thereafter, we discuss in Sect. 4 how ESMs can be utilized to perform incremental development. The contribution of the ESMs to reduce the complexity of model checking is pointed out in Sect. 5 while Sect. 6 introduces a tool to generate ESMs automatically from activities. We close with a discussion of related work and concluding remarks.

2 Activities and Streaming Parameter Nodes

Figure 2 shows an activity which sends SMS messages to mobile phone users. The surrounding system passes SMS messages to be sent out via streaming parameter node *send*. The actual sending happens via a web service call to a Parlay X server [13] within action *s*, which refers to a subordinate activity, taken from [6]. Since this invocation takes some time, SMS messages arriving in the meantime via *send* are stored in a buffer variable. In addition, the logic in Fig. 2 takes care of authentication and optional re-sending in case of errors. This activity is part of our library for telecom services provided by the PATS laboratory operated by Telenor [14], further described in [6].

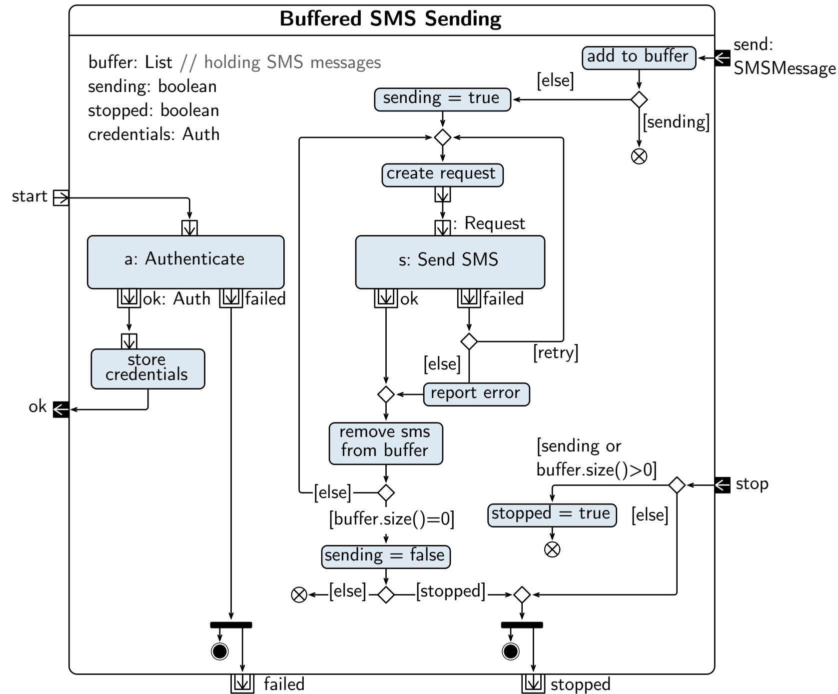

Fig. 2. Activity for Buffered SMS Sending

The activity is started via pin *start*, which invokes call behavior action *a: Authenticate* to retrieve authentication credentials from the Parlay-X server. If this inquiry fails via pin *failed* on *a*, activity *Buffered SMS Sending* terminates via *failed*. If the authentication is successful, the credentials are stored, and a token is emitted via *ok* to signal the surrounding system that SMS messages for sending are accepted from now on. These SMS messages arrive via parameter node *send*, and are added to the buffer. If the activity is currently in the process of sending another SMS, indicated via variable *sending*, the token flow ends. If it is not sending, the flow continues by setting flag *sending* and preparing a sending request, which combines the first SMS in the buffer with the authentication credentials. This request is used to start action *s*. If the sending of the SMS fails, a repetition is possible, depending on guard *retry*, which we do not detail further. If the sending of the SMS was successful or should not be repeated, the SMS is removed from the buffer. In case there are further messages in the list, sending continues with the next message.

To terminate the activity, the surrounding system sends a token via node *stopped*. If the activity is currently sending an SMS message, or the buffer is not yet emptied, only flag *stopped* is set, and the termination is deferred until the buffer is emptied. In the other case, the activity is stopped immediately and a token is emitted via *stopped*.

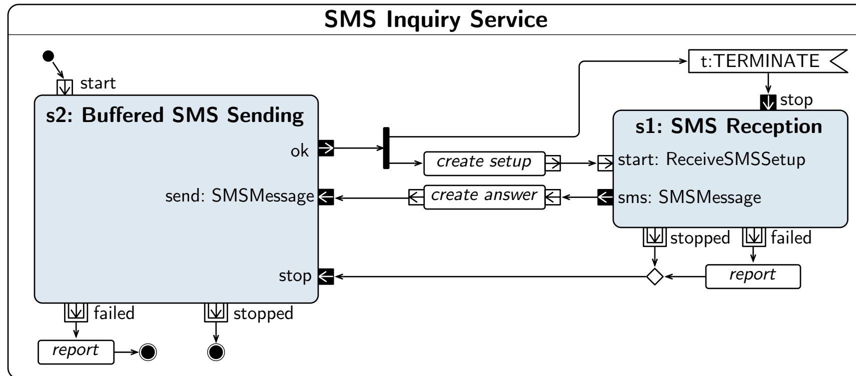

Fig. 3. Example system composing two building blocks *s1* and *s2*

The activity in Fig. 2 uses different types of parameter nodes. The input parameter (⊟) activates an activity and the output parameters (⊟) emit tokens once the activity has terminated. Since the output nodes *failed* and *stopped* are alternatives, they are assigned to different parameter sets, indicated by the additional box. The other parameters *ok*, *send* and *stop*, are *streaming* parameters, here shown as filled boxes (■). They can emit or accept tokens during the execution of an activity, i.e., while it is active.

Streaming parameter nodes enable an interleaving composition in which several call behavior actions may be active, modeling separate functionalities of a system, and may synchronize with each other every now and then. In the systems of our case studies introduced in Sect. 4.1, such interleaving compositions using streaming parameters occur very often. About 65% of all building blocks in our libraries use them.

The activity in Fig. 3 illustrates such an interleaving composition of two subordinate activities, referred to by call behavior actions *s1* and *s2*. The system realizes a simple SMS-based inquiry service, in which mobile phone users may request information such as a weather forecast by sending a certain keyword to a special number. To the right, call behavior action *s1* refers to an activity *SMS Reception* taken from our library. This activity can receive incoming SMS messages that are sent by mobile phone users to a certain number. Call behavior action *s2* refers to the activity for sending out SMS messages described above. With the activity nodes and edges surrounding them, these two building blocks are composed to obtain the complete system specification. When the system starts, a token is emitted by the initial node (●) and action *s2* starts the block for buffered SMS sending by contacting the corresponding web service. In case the startup fails, a token is emitted via *failed* of *s2* and the system is terminated. In case of success, a token is emitted via pin *ok*, which starts sub-activity *s1*. In addition, it places a token into accept signal action *t: TERMINATE*, which receives a token when the system should be terminated. Once users send in SMS messages, they are received by *s1* and a corresponding data object is emitted via

pin *sms*. This SMS is then processed by operation *create answer*. For the keyword *weather*, for example, the current weather forecast is retrieved and wrapped into a new SMS message which is then sent out to the user. When the system is terminated, a token is emitted from *t*, stopping first *s1* and then *s2*.

3 External State Machines – ESMs

An engineer not involved in the design of activities *Buffered SMS Sending* and *SMS Reception* does not know in which exact order parameters have to be passed to or expected from the activity. To construct a sound system, however, this knowledge is necessary. To hide the internal details such as the one from Fig. 2, we use the ESMs. These are UML state machines, stereotyped with ≪esm≫, that refer with their transitions to the activity parameter nodes of the activity they describe. Parameter nodes are referred to as either *triggers* or *effects*, separated by a "/", depending on where a flow originates. The stereotypes and constraints are further detailed in our profile for service engineering [2].

Figure 4 shows the ESM for the buffered SMS sending activity of Fig. 2. It specifies that after the start of the activity via *start*, the activity is in a starting phase, which can result in the termination via *failed*. Since *start* is invoked from the outside, the label declares *start* as trigger, while */failed* points out that the termination is caused by the internals of the activity, perceived by the surrounding context as a spontaneous transition. If, however, the start is successful, *ok* emits a token, and the activity is in its active phase. Within this phase, the activity accepts SMS messages via *send*. To stop the activity, we may in this phase send a token through *stop*. If the block's internal buffer is empty and no SMS messages are left to send out, the stopping happens immediately, and a corresponding token is emitted via the output node *stopped*. This is specified by the transition labeled *stop/stopped*. If there are still SMS messages to send out, the eventual termination of the activity is delayed until all messages are processed, and output node *stopped* emits a token after phase *stopping*. Figure 5 shows the ESM of the SMS Reception activity.

Obviously, for a system to be sound, all activities must actually implement the behavior described by their ESMs, i.e., all ESMs must be true abstractions of their respective activities. For this reason, we defined the formal semantics of

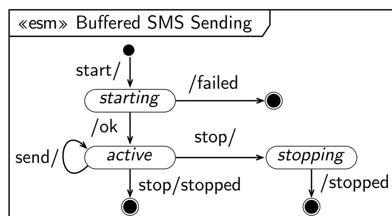

Fig. 4. ESM for Buffered SMS Sending

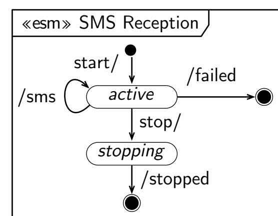

Fig. 5. ESM for SMS Reception

activities using the Temporal Logic of Actions (TLA, [15]), as introduced in [16]. Each activity corresponds to a temporal logic specification A_i, describing all its possible behaviors by a set of actions. An ESM is expressed by a specification E_i. Since an implementation relation in TLA corresponds to logical implication, for any building block i, $A_i \Rightarrow E_i$ must hold. This formula means that each action of an activity maps to a compatible action of the ESM, or the ESM is not involved in the action.

To ensure this sound relation between an activity and its ESM, our tools support two strategies, named *encapsulation* and *refinement*:

- **Encapsulation of existing Activities by an ESM.** Following this development strategy, an existing activity A_x solving a certain problem x is encapsulated by an ESM E_x, so that $A_x \Rightarrow E_x$ holds. In Sect. 6, we describe a tool to generate the ESM from a given activity.
- **Refinement of a given ESM by an Activity.** In this development strategy, a building block to contribute some function y is first described by its externally visible behavior E_y. Since A_y is more detailed than E_y, it can in general not be automatically derived from E_y and is a manual engineering step. However, the necessary refinement relation that must hold can be ensured by an automated verification based on model-checking. We have implemented this by our tools presented in [8,17,18].

We should note that users of our tools are not required to work with temporal formulas. Feedback about the consistency of a specification is given on the level of activities, as explained later. TLA is therefore merely used as a reasoning instrument to ensure that the method and tools are sound.

4 Incremental Development with ESMs

The encapsulation of activities in ESMs facilitates an incremental development style, in which systems can be specified activity by activity, with the ESMs as contracts separating the individual activities from each other. In particular, two styles are enabled by the previously introduced strategies of encapsulation and refinement:

- The strategy of *encapsulation* supports a bottom-up development style, depicted in Fig. 6, in which an ESM is generated for an activity A_x, which can be composed in an enclosing activity S_x together with other activities.
- Vice versa, the strategy of *refinement*, in which an ESM Ey is used to initially specify the abstracted behavior of an activity A_y, supports a top-down development style, illustrated in Fig. 7. Here a higher level specification S_y is developed first, and the subordinate activity A_y is in a first step only described by its ESM E_y. Later on, E_y can be implemented even by a developer unaware of S_y since its expected behavior is described by E_y.

Systems usually have several decomposition levels, with each level corresponding to an activity referred to by call behavior actions. Throughout the development

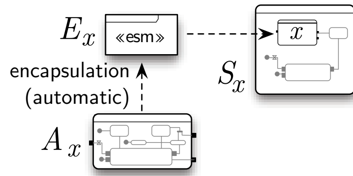

Fig. 6. Encapsulating an activity

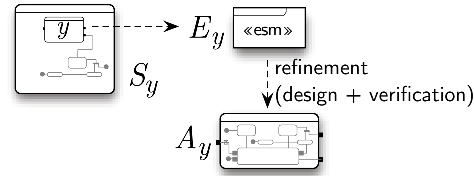

Fig. 7. Refining an ESM

of a system, both styles may be combined: An activity developed bottom-up may at some level be composed with one that is to be developed top-down, and an initial top-down design of an application may be refined until a level is reached where existing activities can be used and encapsulated. If an activity is useful in a number of applications, it can be stored in a library and reused later in other systems.

4.1 Case Studies

To evaluate the impact of the presented specification technique with streaming parameters and ESMs, we conducted a number of case studies, covering several domains:

Web Services. For the orchestration of web services, we demonstrated in [6] how WSDL descriptions can be imported automatically as activities. Each web service operation can be invoked by corresponding streaming parameter nodes. The ESMs ensure that these operations are invoked in a sensible order only.

Embedded Systems. In [4] we composed a sensor network from reusable building blocks. A complete leader election protocol is contributed by one single activity, encapsulated by an ESM. The system was automatically implemented on Sun SPOTs for embedded Java [19].

Mobile Services. In [17] we developed a mobile, location-aware application, in which users solve tasks depending on their current location. This system is used within the FABULA project for mobile learning platforms [20]; the developed activities are also usable in other application areas.

Home Automation. Within the project ISIS (Infrastructure for Integrated Services), we develop solutions for the domain of home automation together with our project partner Telenor. In [7], we demonstrate the composition of a remote fire alarm, in which most parts are reused from libraries.

Trust Management. [5] presents a number of activities encapsulated by ESMs for the domain of trust management.

4.2 Libraries of Reusable Building Blocks

The ESMs act as behavioral interfaces [21] that can be used to separate the work of different developers. When a new activity is introduced providing some

functionality, only its ESM needs to be known in order to use it correctly in an enclosing activity. This facilitates the provision of domain-specific libraries by experts. With the library for trust management [5], for instance, also non-experts in trust management can provide trust-based functions in systems. Due to the ESMs, the correct invocation of these activities is ensured, which guarantees that the trust-based functions are applied correctly.

To determine the degree of reuse enabled by the activities encapsulated by ESMs, we use the reuse proportion R described in [22]. This metric represents the proportion of reused code lines to overall code lines. For the application to UML models, we count the number n of nodes and edges in an activity instead. For a system specification consisting of many activities, each n is then either added to n_{reused} or $n_{specific}$, depending on if it is reused from a library or developed specifically for the application. The resulting reuse proportions R in percent for each system from our case studies is shown in Table 1. The numbers indicate that, in average, 71% of a system specification are contributed by reusable blocks from libraries.

Table 1. Reuse proportions R in percent

	n_{reused}	$n_{specific}$	R
Trusted Auction System [5]	228	76	**75**%
Telecom Web Service System [6]	334	89	**79**%
Treasure Hunt System [17]	131	73	**64**%
Mobile Alarm System [7]	145	70	**68**%
Embedded Sensor System [4]	144	75	**71**%

To use the words of Wills and D'Souza in [23], the reuse enabled by ESMs is a "good one," since it goes beyond simple copy-paste of specification fragments. This is also characterized as compositional *black-box* or *verbatim* reuse [24]. In Sect. 5 we will point out that the reuse holds also for verification purposes, i.e., an activity once verified does not have to be verified again when is is reused. This implies that a reuse proportion of 71% implies real gains in productivity.

The ESMs also serve as an effective guard for changes: Any activity may be modified arbitrarily without affecting the soundness of the system as long as it complies with its original ESM. From a practical point of view, this means we can update and improve the internal realization of a building block in a library without affecting applications using it. Illegal changes harming the ESM are detected by our automatic analysis tools.

5 Incremental Verification with ESMs

Due to the formal definition of the activity semantics based on temporal logic in [16], we can use the technique of model checking for the analysis of specifications. The examples in [8,17] demonstrate how this process can be performed

automatically on UML activities in order to check numerous properties that should hold for any application, like the freedom of deadlocks or bounded communication queues. Problems identified are presented in the form of animations and annotations within the diagrams, as demonstrated in [7], so that engineers do not require a formal background to assure the quality of their models.

A well-known challenge of model checking is the problem of state explosion [25], i.e., that realistic systems often have so many reachable states that a complete analysis cannot be handled within acceptable time. By using ESMs, however, we can verify systems *incrementally*, since each activity is analyzed *separately*. When an activity is model checked, all its subordinate activities referred to by call behavior actions are represented by their respective ESMs. This reduces the number of states to be checked significantly, since the ESMs have usually much less states as they are more abstract than the activities they encapsulate. To achieve that, our model checker verifies two properties for each activity:

(i) The activity has always to comply with its own ESM, i.e., $A_x \Rightarrow E_x$ as mentioned in Sect. 3 holds.
(ii) An activity must always fulfill the ESMs of its subordinate activities.

Formally, a system S using activity A_x is described by $S \triangleq A_x \wedge N$, with N as the behavior of the surrounding context of A_x (see [16]). To prove a property I during the analysis, $P_A \triangleq S \Rightarrow \Box I$ must hold.[1] Using the ESMs instead, the model checker verifies the less complex proof $P_E \triangleq E_x \wedge N \Rightarrow \Box I$. Since $P_E \wedge (A_x \Rightarrow E_x) \Rightarrow P_A$ holds trivially and (i) holds, the replacement of the activities by their ESMs is formally correct. (See also [2].)

The degree of reduction of the size of the state space is discussed below. Further, when an activity is reused, the analysis effort spent will be reused as well. We assume that the designer of a building block only adds an activity to a library after it passed the analysis and does not contain any errors. Thus, other engineers may simply apply the building block without the need to check the correctness of it's internal behavior again. They only have to prove that the environment of the block complies with its ESM.

It is also beneficial for the human developers that the analysis is focused on one activity at a time: Once an erroneous situation is identified by the model checker, the underlying problem is typically easier to understand and solve when only a single activity has to be understood. This makes it also possible to study intricate synchronization problems isolation, as demonstrated in [8].

5.1 Scalability and Reduction of State Space

To make a point in case, we consider a simple example from the domain of Grid technology. These systems stand out for their high number of processes running in parallel. Here, each combination of the local process states forms a

[1] In temporal logic, $\Box$ is the “always” operator stating that a property holds in all states of a system description.

unique system state to be checked separately. Formally, if a system consists of p independent processes and each process may reach s different process states, the overall system contains up to s^p many different system states. If, however, we model each process by a separate activity, this will comprise only s different states. The ESMs of the activities typically contain only two or three states modeling whether the process is either *idle*, *active*, or *terminated*. Thus the overall system model encompassing p call behavior actions for each of the processes affords only two or three states since all processes can be started and terminated at once. Thus, if the sub-activities differ for each process, we have to check altogether only $p \cdot s + 3$ different states. In the case that all processes are identical and we can model them by the same activity, the effort is even reduced to $s + 3$ reachable states since this activity has to be verified only once. So, we can reduce exponential complexity with respect to the number of processes and polynomial complexity with regard to the number of process states to linear complexity.

Also for systems with less parallel behavior than the one sketched above, the reduction of states to be proven is still significant. For instance, the trusted auction system presented in [5] has in total 957 distinct reachable states when the global specification is explored, although it only models two communicating parties and has only three decomposition levels. When we use the ESMs, however, and analyze each activity of the system separately, the state spaces to explore have only a size of 38, 63, 5, 6, 54, and 50 states. Thus, even for such a relatively small system, we could reduce the maximum number of states to be checked in one single run from 957 to 63. The fact that four of these six blocks, including the largest one, were taken from our libraries and were already verified, reduced the effective effort even more.

6 Automatic Generation of ESMs from Activities

When designing an activity, the designer needs to make some assumptions about the environment. To describe these, the activity to encapsulate is placed within a minimal environment. In our editor, such an environment is part of a building block, since it is helpful to illustrate a building blocks usage. Figure 8 shows an environment for the buffered SMS sending: repeated SMS sending by the surrounding system is represented by a periodic timer, and the termination is triggered by a timer. Once the activity is instantiated in its context, the construction of the ESM is completely automated and consists of the following steps:

1. Following the semantics defined in [16], the discrete action steps of the activity within its minimal environment are generated using the tool described in [26]. The state space exploration starts then with the initial marking, in which all initial nodes hold one token. From this initial state, all reachable states are computed by executing all enabled activity steps. As a result, we obtain the state space graph G_x, with the reachable states as nodes and the executed activity steps as edges. The state space during this analysis is limited, since all call behavior actions within the activity to encapsulate are abstracted by their respective ESMs.

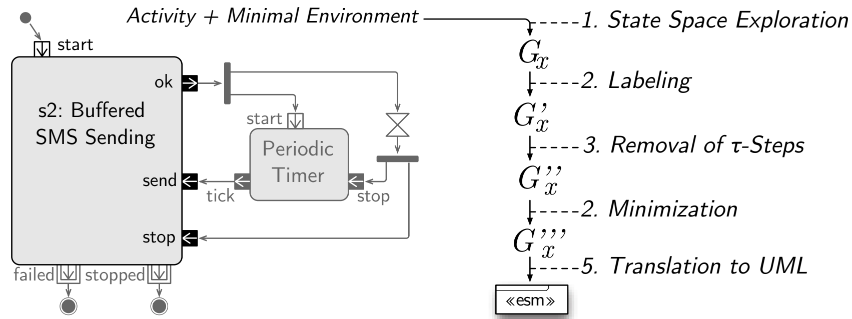

Fig. 8. Illustration of the steps for the automatic encapsulation

2. For each step in the state space, we analyze which parameter nodes of the activity to encapsulate are passed, and assign a corresponding label to the step. If no parameter node has been passed, the step is labelled with τ.
3. The τ-steps do not contribute to the visible behavior expressed by an ESM and therefore removed. For that, every pair of states that is connected by a τ-step is merged, and the τ-step is deleted.
4. After the removal of τ-steps, the resulting graph is minimized using the algorithm for state machine minimization described by Holzmann in [27].
5. From the resulting minimized graph, the UML representation in form of the ESM is constructed. The initial graph state is represented by an initial pseudo state. Each remaining graph state is represented by a UML state, resp. final state if the graph state has no outgoing steps.[2] For each graph step, an ESM transition referring to the corresponding activity parameter nodes is added.

We implemented the algorithm as an Eclipse plug-in, integrated with our modeling tool Arctis [7,28], using the UML repository of the Eclipse Modeling Project [29]. So far, we have used it on over 200 of our activities to encapsulate them by ESMs.

The implication relation between the ESM and the activity is ensured *by construction*, due to the layout of the algorithm. Formally, this can be verified by a refinement proof $A_x \Rightarrow E_x$ in temporal logic. The necessary refinement mapping (see [30]) can be obtained from the algorithm, by observing which states are merged during τ-step removal and minimization. For the Buffered SMS Sending example, we verified this refinement using the model checker TLC [31].

7 Related Work

There exists a number of language constructs to describe the visible behavior of components at distinct interaction points. ROOM [32], for instance, used protocols to define the ordering of signals transmitted by actors. The UML 2.x

[2] The algorithm assigns generic names to the states, which can be renamed in an optional, manual step.

standard proposes *protocol state machines* to define the allowed invocation sequences for operations on an object. Mencel [33] extends these descriptions by *port state machines*, to handle also nested calls and dependencies between required and provided interfaces. For the derivation of visible component behavior, Floch describes in [34] an algorithm that projects the observable behavior (i.e., the transmission of signals) of SDL processes towards specific gates. This work has been adapted in [35] for UML state machines. Our work, in contrast, handles the encapsulation of behavior on the level of activities; components and state machines are generated in an automated process, as described in [10]. The interfaces derived in [34] describe the transmission of signals *between* components. ESMs describe interfaces of activities that are composed *within* components, and do not imply signal transmissions. Rather, two activity flows connected via ESMs can be implemented by the same state machine transition.

Formally, the encapsulation of activities resembles the work of Kellomäkki and Mikkonen [36], who use the DisCo language [37] to capture specifications that are reusable solutions to problems. To reuse solutions in an application, they show that it suffices to integrate a more abstract template, and that properties proven for the solution are maintained when the template is applied. While this work is also based on temporal logic and uses refinement relations, it does not provide a mapping to UML as our work does.

8 Concluding Remarks

The streaming parameters of UML 2.x activities are a useful concept to enable the composition of systems from building blocks expressed by activities. From all building blocks involved in the case studies presented in Sect. 4.1, about two third make use of streaming parameters, so that activities may be composed in an interleaving manner. This enables that related functions may be offered as coherent, self-contained building blocks in the form of activities, but still can synchronize control and data flows with other parts of the system throughout their execution. To abstract from inner details and focus on the visible behavior at the streaming pins of an activity, we proposed the concept of ESMs, and described and implemented an algorithm to construct it. We have shown how this facilitates the provision of libraries, and how the compositional verification of systems is made possible by using ESMs as an abstraction mechanism. In addition, since, once consistent, a building block is encapsulated, an incremental development style is possible, in which systems can be designed, verified and composed *block by block*.

References

1. Object Management Group. Unified Modeling Language: Superstructure, version 2.0, formal/2005-07-05 (2005)
2. Kraemer, F.A.: Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks. PhD thesis, Norwegian University of Science and Technology (2008)

3. Kraemer, F.A., Herrmann, P.: Service Specification by Composition of Collaborations — An Example. In: Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology), pp. 129–133. IEEE Computer Society, Los Alamitos (2006)
4. Kraemer, F.A., Slåtten, V., Herrmann, P.: Model-Driven Construction of Embedded Applications based on Reusable Building Blocks – An Example. In: Bilgic, A., Gotzhein, R., Reed, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 1–19. Springer, Heidelberg (2009)
5. Herrmann, P., Kraemer, F.A.: Design of Trusted Systems with Reusable Collaboration Models. In: Etalle, S., Marsh, S. (eds.) Trust Management. IFIP International Federation for Information Processing, vol. 238, pp. 317–332. Springer, Heidelberg (2007)
6. Kraemer, F.A., Samset, H., Bræk, R.: An Automated Method for Web Service Orchestration based on Reusable Building Blocks. In: Proceedings of the 7th International IEEE Conference on Web Services (ICWS), pp. 262–270. IEEE Computer Society, Los Alamitos (2009)
7. Kraemer, F.A., Bræk, R., Herrmann, P.: Compositional Service Engineering with Arctis. Telektronikk, vol. 1.2009 (2009)
8. Kraemer, F.A., Slåtten, V., Herrmann, P.: Engineering Support for UML Activities by Automated Model-Checking — An Example. In: Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques, RISE (2007)
9. Kraemer, F.A., Bræk, R., Herrmann, P.: Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 166–185. Springer, Heidelberg (2007)
10. Kraemer, F.A., Herrmann, P.: Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In: Ehring, K., Giese, H. (eds.) Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007). Electronic Communications of the EASST, vol. 7. EASST (2007)
11. Kraemer, F.A.: Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart (2003)
12. Merha, B.T.: Code Generation for Executable State Machines on Embedded Java Devices. Project Thesis, Norwegian University of Science and Technology, Trondheim, Norway (2008)
13. Parlay Group. Parlay X Web Services Specification, Version 2.1 - Short Messaging, `http://www.parlay.org/en/specifications/pxws.asp`
14. PATS Lab Website, `http://www.pats.no`
15. Lamport, L.: Specifying Systems. Addison-Wesley, Reading (2002)
16. Kraemer, F.A., Herrmann, P.: Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In: Networking and Electronic Commerce Research Conference 2007 (NAEC 2007), pp. 194–220. ATSMA Inc. (2007)
17. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software (to appear, 2009)
18. Slåtten, V.: Automatic Detection and Correction of Flaws in Service Specifications. Master's thesis, Norwegian University of Science and Technology (2008)
19. `http://www.sunspotworld.com`

20. Kathayat, S.B., Bræk, B.: Platform Support for Situated Collaborative Learning. In: Proceedings of the 2009 International Conference on Mobile, Hybrid, and Online Learning, Cancun, Mexico, pp. 53–60. IEEE Press, Los Alamitos (2009)
21. Beugnard, A., Jézéquel, J.-M., Noël, P., Watkins, D.: Making Components Contract Aware. IEEE Computer 32(7), 38–45 (1999)
22. Gaffney, J.E., Durek, T.A.: Software Reuse – Key to Enhanced Productivity: Some Quantitative Models. Information and Software Technology 31(5), 258–267 (1989)
23. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML: the Catalysis Approach. Addison-Wesley, Reading (1999)
24. Frakes, W., Terry, C.: Software Reuse: Metrics and Models. ACM Computing Surveys 28(2), 415–435 (1996)
25. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
26. Slåtten, V.: Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, Norwegian University of Science and Technology, Trondheim, Norway (2007)
27. Holzmann, G.: Design and Validation of Computer Protocols. Prentice Hall Software Series. Prentice-Hall, Englewood Cliffs (1991)
28. Arctis Website, `http://arctis.item.ntnu.no`
29. Eclipse Modeling Project, `http://www.eclipse.org/modeling`
30. Abadi, M., Lamport, L.: The Existence of Refinement Mappings. Theoretical Computer Science 82(2), 253–284 (1991)
31. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA^+ Specifications. In: Pierre, L., Kropf, T. (eds.) Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99), LNCS, vol. 1703, pp 54–66. Springer, Heidelberg (1999)
32. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., New York (1994)
33. Mencl, V.: Specifying Component Behavior with Port State Machines. Electronic Notes in Theoretical Computer Science 101, 129–153 (2004)
34. Floch, J.: Towards Plug-and-Play Services: Design and Validation using Roles. PhD thesis, Norwegian University of Science and Technology (2003)
35. SIMS Project Website, `http://www.ist-sims.org`
36. Kellomäki, P., Mikkonen, T.: Design Templates for Collective Behavior. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 277–295. Springer, Heidelberg (2000)
37. Järvinen, H.-M., Kurki-Suonio, R., Sakkinen, M., Systä, K.: Object-Oriented Specification of Reactive Systems. In: Proceedings of the 12th International Conference on Software Engineering, pp. 63–71. IEEE Computer Society Press, Los Alamitos (1990)