Parametric Statecharts: Designing Flexible IoT Apps

Deploying Android m-Health Apps in Dynamic Smart-homes

Roopak Sinha
IT & Software Engineering
Auckland University of Technology
Auckland, New Zealand
roopak.sinha@aut.ac.nz

Ayush Narula
IT & Software Engineering
Auckland University of Technology
Auckland, New Zealand
developeron29@gmail.com

John Grundy
Deakin University
Geelong, Australia
School of Information Technology
j.grundy@deakin.edu.au

ABSTRACT

Mobile apps can integrate sensors and actuators in Internet-of-Things systems to achieve novel and diverse functionalities, for example self-management and monitoring functions to help patients manage a large number of health conditions within their (smart-) homes. However, each smart-home may contain a different and often dynamic sensor-actuator configuration and it is undesirable to write new code for every new installation or change. Statecharts present an appropriate formal and visual design model to design apps and support automatic code generation. However, these designs assume a specific and static sensor-actuator configuration. We propose *parametric* statecharts, an extension to statecharts that can be automatically customised to a dynamic smart-home's configuration. We develop a translator to convert parametric statecharts into standard statecharts customised to a given system configuration, and then a custom compiler to generate Android code. Experimental results confirm the flexibility of the proposed approach.

CCS Concepts

•**Software and its engineering** → *Syntax; Source code generation; Application specific development environments;*

Keywords

IoT, statecharts, m-health apps, smart-homes, mobile apps, automatic code generation, design

1. INTRODUCTION

Internet-of-Things (IoT) architectures have opened up the potential for software apps to orchestrate a diverse range of devices and achieve novel functionalities like home automation, remote monitoring, and health care. In this paper, we consider the example of mobile-Health (m-Health) applications help patients manage a number of health conditions at home at low cost. Such applications, such as rehabilitation plans, are currently delivered as paper-based instructions, and the rate and accuracy of their adherence can vary widely [17]. Using a patient's mobile phone to implement these apps within a smart-home can potentially help increase their effectiveness.

Technically, running rehabilitation m-Health apps on smart-phones is seemingly quite easy, given the ability of even low-end smart-phones to host apps that can issue reminders, connect to the Internet, interact with patients, and even orchestrate sensors and actuators within a smart-home. However, there are some key challenges that remain unaddressed. Firstly, we cannot expect to write custom apps for each patient and their smart-home configuration (the set of sensors and actuators they have). Secondly, m-Health apps tend to be highly control-dominated [8] and manually changing deeply nested if-else code when the overall app functionality changes can easily introduce bugs. Thirdly, while apps can easily interface with available sensors and actuators in a given IoT system such as a smart-home, the configuration of a user's smart-home may not be static. In fact, sensors may go down, and new devices may be added. An app must then adapt to these changes, which is a time-consuming exercise considering the coding effort and expertise required to handle each change scenario manually.

A number of solutions address some of these problems in isolation. CodeBlue [6], a remote patient tracking and monitoring tool, uses sensors to capture and transmit patient related data to clinicians. Odin [24] is an intermediate platform that can be used to develop systems for remote health services more easily. None of the existing solutions provides the flexibility and level of automation required for deploying flexible apps in dynamic smart environments. Sensor fusion and aggregation techniques can potentially help [21], but require an additional service layer to maintain the registry of devices and fusion algorithms.

In this paper, we propose a solution to design and automatically deploy applications in IoT systems with dynamic configurations. We choose a formal model that is useful for visual design and yet amenable to automatic code generation. We use *statecharts* [1], a popular formal model used in a range of applications areas. We develop an automatic tool that can read a statechart and generate Android code for a m-Health apps. However, statecharts are too static to solve the final problem of handling changing sensor-actuator configurations. We therefore extend statecharts to *parametric statecharts* to help design apps more flexibly. Parametric statecharts models are independent of configurations but can

ACSC Melbourne, Australia

DOI: 10.475/123_4

later be automatically translated to traditional statecharts models customised to a specific smart-home configuration. Hence, an app can be easily deployed to different or changing configurations. Parametric statecharts can just as easily be used in other application areas, such as home automation, where software must flexibly interact with dynamic sets of sensors and actuators.

The primary contributions of this research are:

1. The formulating of parametric statecharts to capture the inherent flexibility of software that must interact with dynamic sensor-actuator configurations in IoT systems.

2. A tool to generate highly customised and contextualised mobile code for apps in smart environments.

3. Demonstrating an application of our approach using a case study concerning m-Health apps in smart-homes.

The next section presents motivation and related works and a rationale for choosing statecharts as the model of choice for this work. Subsequent sections present the details of generating mobile code from statecharts (Sec. 4), parametric statecharts formulation and conversion to traditional statecharts (Sec. 5), and preliminary experimental results from a variety of m-Health case studies (Sec. 6). Concluding remarks appear in Sec. 7.

2. MOTIVATION

A fall detection plan: m-Health apps can range simple medication and exercise reminder schedules, to managing potentially life threatening situations such as real-time monitoring and resolution of falls in old or ill patients [16]. We choose a simplified fall assessment and detection plan, which can be listed as a sequence of following steps:

1. If the patient is awake, monitor their movement or request feedback from the patient at regular intervals.

2. If the patient is moving or standing, monitor any sudden change of altitude.

3. If the patient has been moving or standing for an extended period of time, remind them to sit down.

4. If a fall occurs, report it immediately and monitor patient's vital signs.

5. If the fall is not serious, the patient can override the alerts.

The steps above are typical of m-Health apps - they tend to be repeating sequences of decisions. In smart-environments, a fall assessment and detection app can use a wide variety of sensors, ranging from image processing via video surveillance [16] to custom on-body sensors [25]. A crucial challenge is to take a generic design and customise it to the kinds of sensors available in a patient's smart-home. A further challenge is to re-use an existing design in case the available sensors and actuators (alert mechanisms) change. We focus

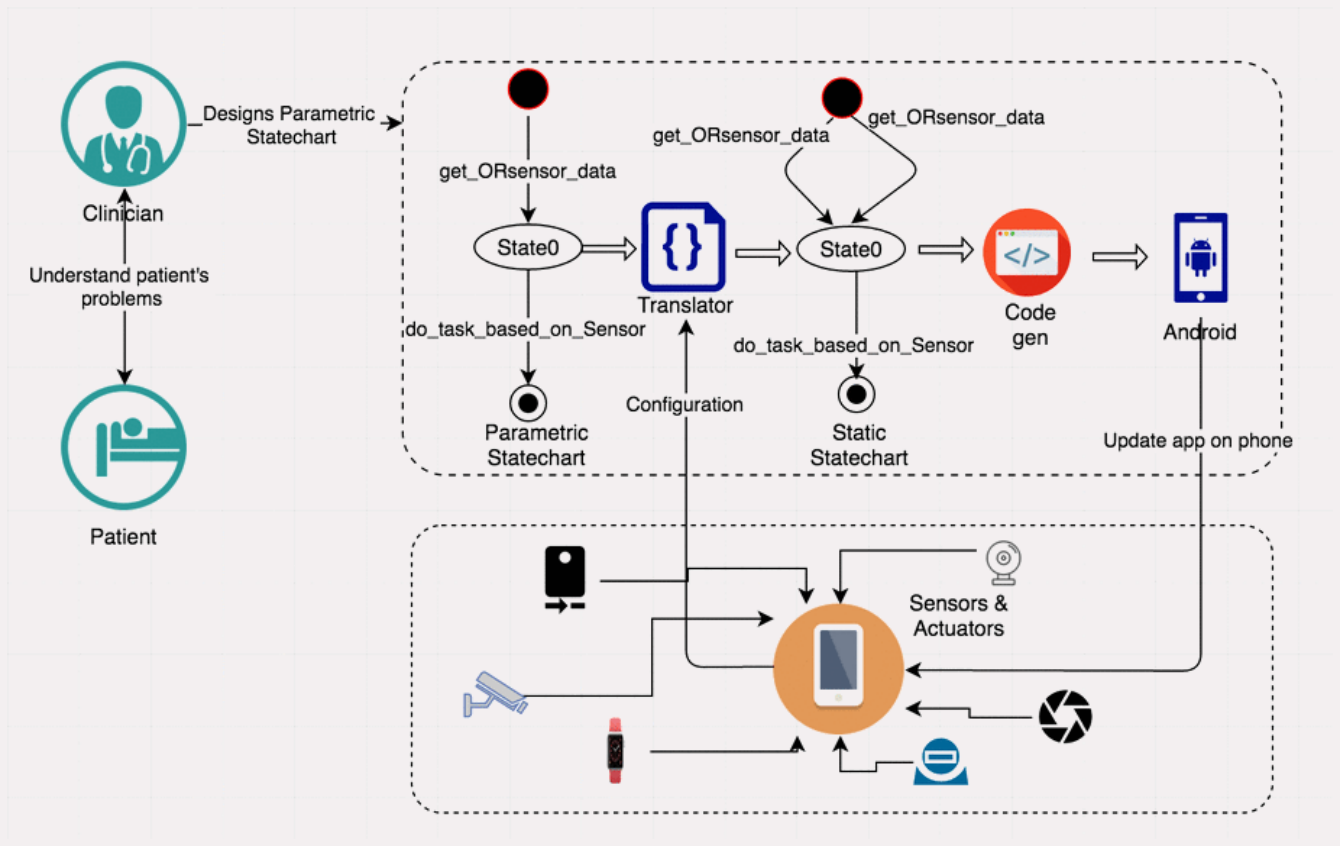


Figure 1: Overview of the proposed solution

on these challenges and present a solution to flexibly deliver m-Health apps in smart-homes.

Fig. 1 shows an overview of our approach. The app can be designed visually using a parametric statechart. This parametric statechart is stored in a database hosted on a server. The user logs-in to a shell application on their phone, which sends the configuration of the smart-home (features and details of sensors and actuators) to the server. The server then finds the relevant design, and generates first a static statechart from the parametric statechart. This static statechart is customised to the user's smart-home configuration and includes the code to read/write to/from the the sensors/actuators in the smart-home. The static model is then compiled into Android code, which is installed as an update to the shell-application. Any subsequent change in the configuration is sent to the server, which can re-generate code and send it through as another update to the application.

3. RELATED WORK

IoT offers a dynamic platform where well-designed software can achieve novel functionalities. The focus of our chosen case study is the deployment of m-Health apps in dynamic smart environments. m-Health apps, such as rehabilitation plans help achieve goals that align with an individual's work and day schedules, specific conditions they need to manage, and their physical and mental capacity [7]. m-Health apps help address issues regarding the unavailability of real-time patient-related data during home-care [15], reinforcing healthier habits [23], and high costs of traditional healthcare [12]. A carefully designed solution can improve several other key quality attributes at the individual, organizational and contextual levels [5].

Choosing the right model for designing apps for IoT systems such as smart-homes is a critical decision. The model must be easy to use, preferably visual, and use language that can be understood by domain-experts. It must also be formalised to some extent, so that we can translate it into mobile code without further manual effort. There is a plethora of modelling languages used to design and develop workflows for various types of systems. Since most IoT apps integrate and orchestrate a diverse range of devices, a majority of them tends to be control-dominated. Such apps contain sequences or nestings of decision making, and that is often the case with m-Health apps. Hence, we rule out data-driven formal models and constrain ourselves to the class of control-dominated IoT apps.

Finite state machines are visual and naming conventions for states and events triggering transitions between states can easily be adapted to domain-specific constraints. They are also conducive to code generation. However, they lack the concurrency that is required in orchestrating several devices in IoT systems. We therefore choose statecharts, which extend finite state machines to provide hierarchy and concurrency support [9]. Statecharts have been previously used in healthcare for developing graphical diagrams representing work flows in surgical care operations [20]. They have also been used to build mobile software [4] and adopted in many popular design frameworks, such as UML statecharts [10] with several works on creating statecharts from other modelling languages [26].

The advantage of using statecharts over other modelling languages in m-Health applications comes from the way these applications are specified. These apps typically mention specific states (awake, asleep, time of day etc.) and events that trigger transitions between these states. statecharts can also isolate specific functionalities, such as the generation of an event by fusing information from multiple sensors, and have the ability to visually model control flow. We do not target the creation of a clinician-friendly template language for designing statecharts-based m-Health app designs in this paper, but given the compatibilities listed above, the creation of such a template language looks feasible.

There are tens if not hundreds of variants of statecharts. These variants differ in their execution semantics which deal with issues such as inter-level transitions, lifetime of events, determinism, etc. [22]. We use the Yakindu tool [13] to model statecharts in this paper. Yakindu supports a cyclic, and hence deterministic execution semantics of statecharts, which could be useful in the m-Health context. Yakindu also supports a Java compiler which is more compatible for Android code generation targeted in our case study. However, we aim for a more general approach, where any statecharts variant can be used as per domain-specific preferences.

While statecharts adequately capture the logic of IoT apps, they are not sufficient for capturing the dynamic nature of the devices in an IoT system. Each system, such as a smart-home, contains a different set of sensors and actuators that an app may exploit to achieve a specific functionality. Using a smart-home's sensor-actuator network or its configuration can help m-Health apps collect data and broadcast alerts automatically. Unfortunately, very little work exists in the area of allowing mobile apps to interact with dynamic configurations. In [2], a mobile assistance system that could be personalized and is context-aware for prevention and rehabilitation of cardiovascular diseases is proposed. The system adapts itself based on a patient's ECG records. While this is somewhat similar to the goals of this research, it cannot adjust to dynamic changes in smart environments, such as when a sensor goes offline. Unfortunately, even traditional statecharts based designs are too static for these cases, as generating mobile code from them requires explicit information about each sensor and actuator.

It is possible to use a sensor-fusion approach [19] to allow an app to discover and use a smart-home configuration. This service would run on the operating system and the app can make service-calls to interact with the smart-home. However, this poses two problems. Firstly, the sensor-fusion required for an app might be closely intertwined into its control-flow, as is the case for many m-Health apps. Secondly, the availability of an additional sensor-fusion service becomes a pre-requisite for the app being deployed, and this may limit the number of platforms where the app can be installed. Controlling sensor-fusion activities within the app can also ensure smaller code and possibly faster performance.

Our research uses statecharts for modelling and automatic code generation of flexible IoT apps in general and m-Health apps specifically. We formulate parametric statecharts to infuse more flexibility in IoT app designs. Domain experts can design apps visually using appropriate templates, without having to know details about a user's smart-home. A design can then be automatically customised to a given smart-space configuration and then translated into Android code running on the user's phone. Parametric statecharts can just as easily be used in other application areas, wherever software needs to interact with dynamic sensor networks. Parametric

statecharts do not compete with the many statechart variants. They can simply be unrolled into statecharts that can then be interpreted over any of these semantics.

4. STATECHARTS-BASED APP DESIGN

We begin with a formal definition of statecharts [3].

Definition 1. A statechart is a tuple $SC = \langle S, T, E \rangle$ where S is a set of states, $T \subseteq S \times E \times S$ is a set of transitions where E is a finite set of events. The function $children : S \to \mathcal{P}(S)$ maps states to children states. States have types, described by the function $type : S \to \{BASIC, OR, AND\}$. $BASIC$ states have no children, OR states are in one of their children states at any time, and AND states are in all children states at the same time. It has a root state s_0 which is of type OR and has no parent. Function $default : S \to S$ identifies one of the children of an OR state as its *default* child. The $root \in S$ is the root state of the statechart, which has no parent and is of the type OR.

Fig. 2 shows an abbreviated statechart for the fall assessment and detection plan described in Sec. 2. The root state of the statechart is not shown, but its default child is state $Init$, indicated by the incoming arrow with no source state. The plan moves to the state $Connected$ after performing some initialization and then to state $Idle$. We use the shorthand $Init \xrightarrow{Person.connect} Connected$ to describe a transition from a source state to a destination state. We use $s \xrightarrow{1} s'$ to describe transitions that are always enabled. If the user is awake, the statechart moves to the state $Awake$ to monitor falls. This state is an AND-type state and incorporates an OR-type state called *main region* as its only child. This relationship models the hierarchy between $Awake$ and the state-machine contained within *main region*. In default state $Monitor$ of this embedded state-machine, the plan monitors the user's presence every 200 seconds and moves to state $Find\,location$ when the user is detected to be present in the home. It then checks for falls by computing the rate of change in the user's position as an event $altitude$. $altitude$ is computed by reading an appropriate sensor and translating

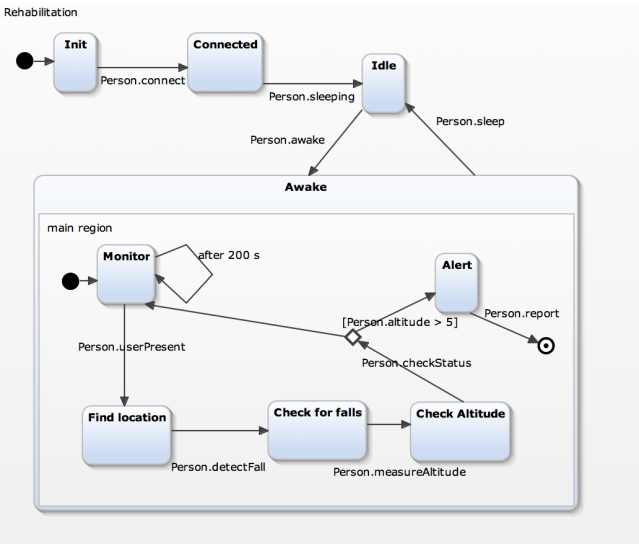


Figure 2: Static statechart plan for fall detection

the raw data into a score ranging from 0 to 10, with a score greater than 5 indicating a potential fall. This internal computations are hidden in the design, as they depend on the set of available sensors. If a fall has occurred, the statechart moves to the state $Alert$ to issue relevant alerts before terminating. Otherwise, it resets back to the state $Monitor$. The user-reset function has been removed to ensure readability.

We use the Yakindu toolkit [13] to draw statecharts-based rehabilitation plans. Hence, in this case study we use the cyclic execution semantics of Yakindu which guarantees deterministic execution [13].

Yakindu allows statecharts to be drawn using event interfaces, and these event interfaces can later be contextualised for specific implementations. E.g., Fig. 2 refers to an interface $Person$, which represents the set of input and output events the statechart exchanges with a user. However, some of these events need to be translated into code that performs input/output operations. E.g., the event $altitude$ relates to reading information from an appropriate sensor and then carrying out further internal computations. We hide these details from the designer, and instead provide event interfaces listing available events. These interfaces are managed in an interface library that maps events to any operation sequences that may be required to evaluate them. In case a designer adds newer events, the event interface library will need to be updated in the back-end. However, these updates are expected to be more stable and less frequent, and once in place, can be reused for a range of m-Health apps.

Yakindu also provides a compiler to generate Java code from a statechart. We use this compiler to create generic Java code which is then customised using the backend interface library mapping events in the event interfaces to code that implements them. This code can include interfacing with sensors using Bluetooth or Wifi. This is done using a custom compiler, which also generates Java and XML code that is customised to run in an Android app. This code can then be packaged as an update to a shell application running on the user's phone.

The ability of a domain-expert in designing an app using statecharts depends on their domain. While a designer of a home automation app may be reasonably expected to draw statecharts, a clinician prescribing a rehabilitation plan will not have the skills to draw a statechart. Hence, providing appropriate tool-support as well as developing domain-specific template languages for designing apps are promising future directions of this research.

5. PARAMETRIC STATECHARTS

Statecharts-based app designs cannot cope with differing configurations of sensors and actuators in an underlying IoT system. E.g., two types of sensors can be used for the m-Health app shown in Fig. 2. Presence sensors located in individual rooms can be used to locate where the user is, and altitude sensors signal if a fall has occurred. A smart-home may have multiple presence sensors (one in each room), and similarly have more than one sensor to measure a user's altitude, such as a sensor built into a user's cane or an on-wrist barometric pressure sensor. It is also possible that any of these sensors could go offline at any time, or the user might add new sensors to their home. The activation of *any* of the presence sensors will signal the user's presence in the home, whereas we may want to aggregate *all* the altitude sensor readings to compute the probability of a fall

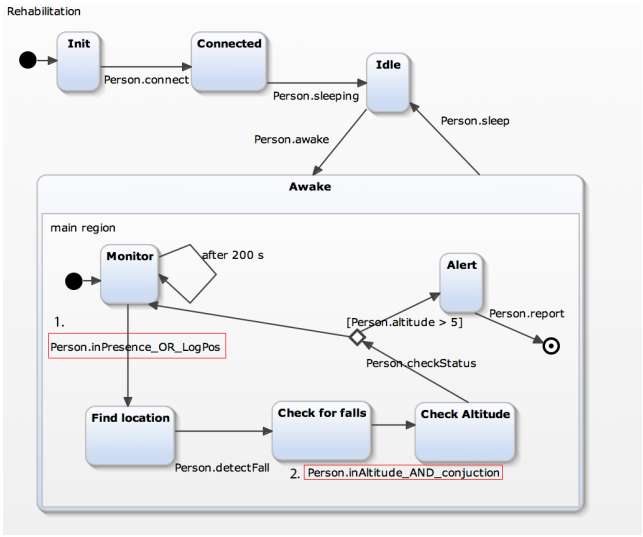


Figure 3: Parametric statechart design for fall detection

happening more accurately [14]. So, not only must an IoT app be aware of multiple sensors or actuators in a given IoT system, it must also know how they are to be used. Hence, we need to add more flexibility to statecharts, which we do using *parametric statecharts*. A parametric statechart is defined as follows.

Definition 2. A parametric statechart $SC = \langle S, T, E \rangle$ is a statechart as defined in Def. 1. The set of events E of a parametric statechart contains a set of interface operation events $HE \subset E$. Each event $e \in HE$ is of the form $feature\_aggr\_endOp$ where $feature$ represents a sensing or actuating feature (e.g. read temperature, etc.), $aggr$ represents how multiple devices supporting the same feature need to be aggregated, and $endOp$ is a user-provided function that processes the data obtained from one or more sensors.

Fig. 3 shows a parametric statechart for the fall-detection app, which is quite similar to the static statechart shown in Fig. 2. It only differs in the naming of the events relating to the presence and altitude sensors, noted in boxes labelled 1 and 2. Event $Person.inPresence\_OR\_LogPos$ corresponds to an event from the interface $Person$ where $inPresence$ relates to the reading of a presence sensor, OR relates to reading *any* sensor in case there are multiple sensors of the same type available. Aggregation operations $aggr$ can be of types OR, AND, or any user-defined operation such as counting aggregations depending on desired sensor fusion requirements. $LogPos$ refers to the logging of the last sensor reading to more accurately track where the user is in their home. Similarly, the event $Person.inAltitude\_AND\_conjunction \in HE$ corresponds to an event from the interface $Person$ where $inAltitude$ relates to reading an altitude sensor, AND relates to reading *all* sensors if there are multiple sensors available, and *conjunction* is a user-defined function that aggregates the results from all sensors to compute the meta-variable *altitude*.

The parametric statechart shown in Fig. 3 is independent of the actual numbers of presence and altitude sensors available. However, to customise the design to work in a given smart-home, we require the precise configuration of the sensor and actuators available. In general, the configuration of an IoT system is defined as follows.

Algorithm 1: CONVERT Converts a parametric statechart to a static statechart

Input: Parametric statechart $SC = \langle S, T, E \rangle$, and IoT system configuration $C = \langle E_{sh}, F \rangle$
Output: Static statechart $SC' = \langle S', T', E' \rangle$

$S' = S$, $T' = T$, $E' = E$;
for *Transition* $t = (s \xrightarrow{e} s') \in S'$ **do**
  if $e \in HE = feature\_aggr\_endOp$ **then**
    if $aggr = OR$ **then**
      Remove t from T';
      for *Each operation* $o \in Operations(feature)$ **do**
        Add transition $s \xrightarrow{o} s'$;
      end
      Add operation $endOp$ to s';
    end
    else if $aggr = AND$ **then**
      Remove t from T';
      Create unique state s_f of type AND and state s_j of type $BASIC$;
      Create transitions $s \xrightarrow{1} s\_f$ and $s\_j \xrightarrow{1} s'$;
      for *Each operation* $o \in Operations(feature)$ **do**
        Create state s_o of type OR as a child of s_f;
        Add transition $s\_f \xrightarrow{o} s\_a$;
        Create states s_o1, s_o2 of type $BASIC$ as children s_o and make s_o1 the default child of s_o;
        Create transitions $s\_o1 \xrightarrow{o} s\_o2$ and Create transition $s\_o2 \xrightarrow{o} s\_j$;
      end
      Add operation $endOp$ to s_j;
    end
  end
end
return C;

Definition 3. An IoT system configuration is defined as the tuple $C = \langle E_{sh}, F \rangle$ where $E_{sh} = E^r_{sh} \uplus E^w_{sh}$ is a set of supported read and write operations and $F = F^r \uplus F^w$ is a set of supported read and write features. Each feature is mapped to a set of operations via the function $Operations : F \to \mathcal{P}(E_{sh})$.

An IoT system configuration links features expected in a parametric statechart app design to operations supported by available sensors and actuators. E.g., the configuration of the smart-home of a user following the fall detection app in Fig. 3 may have three presence sensors and two altitude sensors. We can denote the configuration of this smart-home as $C = \langle E_{sh}, F \rangle$ where $E_{sh} = \{ps1, ps2, ps3, as1, as2\}$. $ps1 \ldots 3$ are read operations for the three presence sensor, and $as1$ and $as2$ are read operations for the two altitude sensors. All these operations are read operations. Also, $F = \{inPresence, inAltitude\}$ as the smart-home allows a mobile app to sense both presence and altitude. Finally, $Operations$ maps these features to the set of available operations, i.e., $Operations(inPresence) = \{ps1, ps2, ps3\}$, and $Operations(inAltitude) = \{as1, as2\}$.

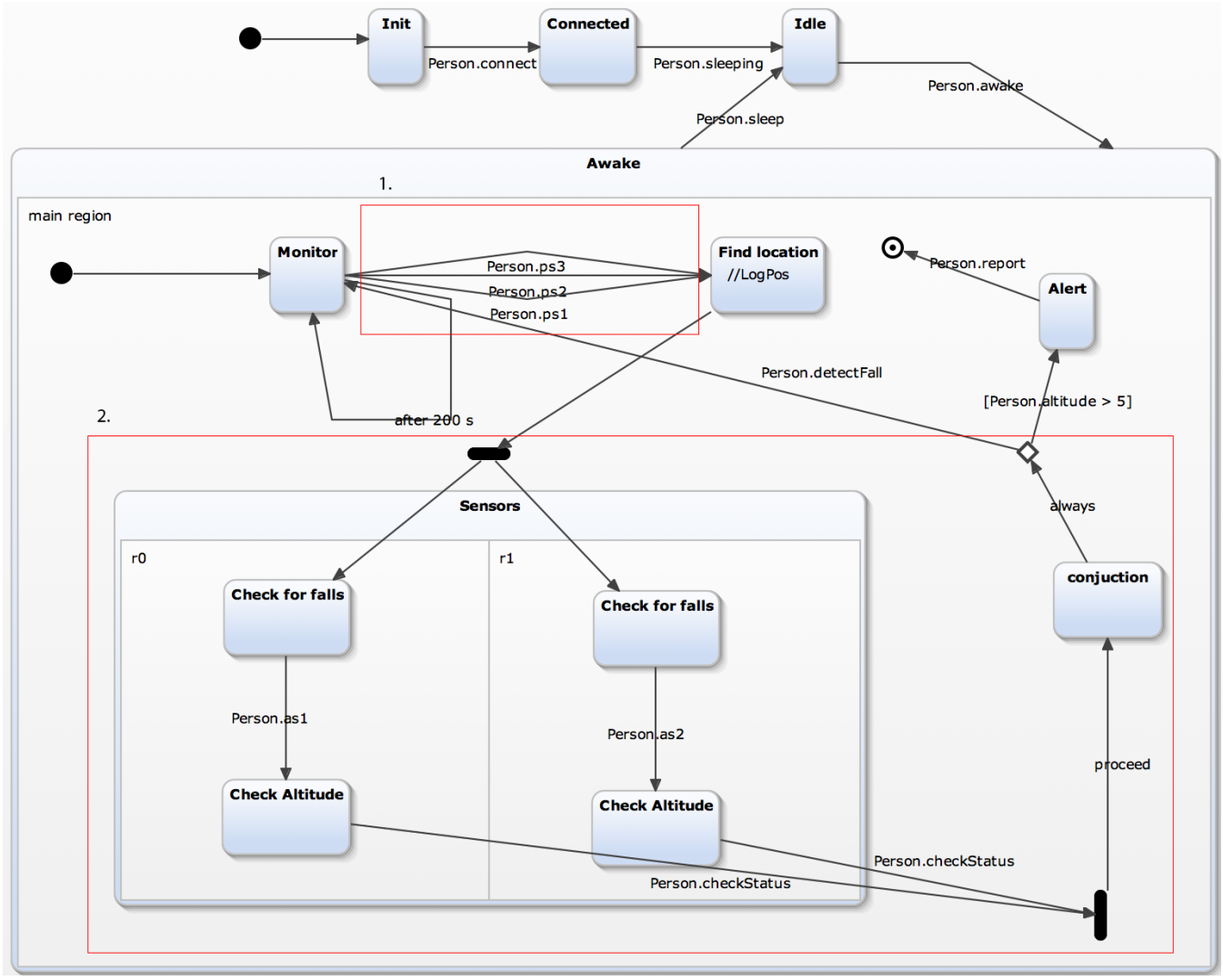


Figure 4: Static statechart generated by Alg. 1 from parametric statechart in Fig.3

Once a system configuration is known, we can generate a static statechart from a parametric statechart using Alg. 1. Fig. 4 shows the statechart generated when the parametric statechart in Fig. 3 is processed with respect to the smart-home configuration C described above. The algorithm processes each transition in the parametric statechart relating to a sensing or actuating event $e \in HE$ (line 2) and replaces it with a customised statechart snippet.

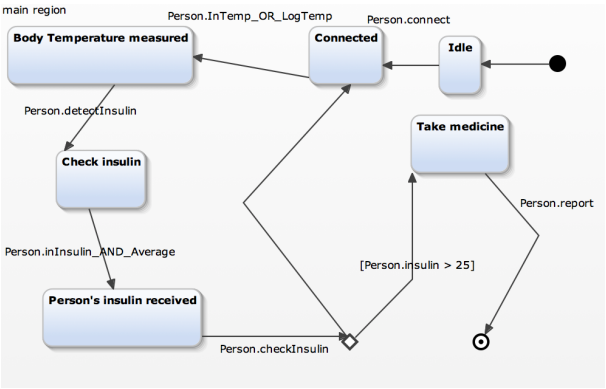
For events that relate to OR type sensors or actuators, the transition is replaced with multiple transitions from the source state to destination state, each labelled by the precise operation the given configuration supports (lines 4–9). E.g., the transition $Monitor \xrightarrow{Person.inPresence\_OR\_LogPos} FindLocation$ shown in Fig. 3 (labelled as box 1) is removed and is replaced by multiple transitions between these states labelled by the operations $Person.ps1 \ldots Person.ps3$. This is labelled as box 1 in Fig.4. The logging function $LogPos$ is moved into state $FindLocation$ and is used to log the last position the user was found to be present in.

For events that relate to AND type sensors or actuators, a transition is replaced by a nested statechart with concurrent state-machines (lines 11–21). Consider the transition

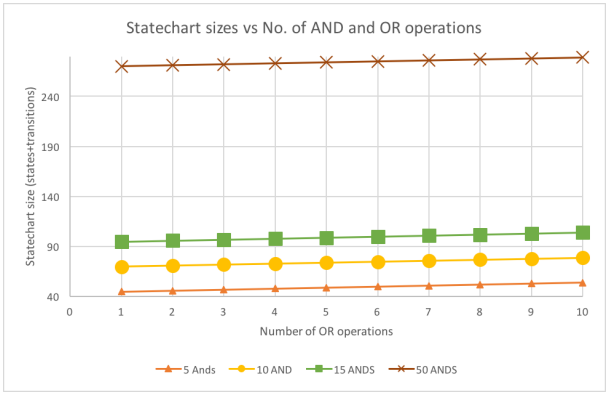
$$Checkforfalls \xrightarrow{Person.inAltitude\_AND\_conjunction} CheckAltitude$$

shown in Fig. 3 (labelled as box 2). Since there are two sensor operations $as1$ and $as2$ that support the $inAltitude$ feature, we replace the transition with the nested statechart snippet in box 2 in Fig. 4. This allows the statechart to read asynchronously from both sensors before moving to the next state $conjunction$, which relates to the user-defined $conjunction$ operation used for computing the meta-variable $altitude$.

Once Alg. 1 terminates, a static statechart is obtained that can be converted into Java code using the process described in Sec. 4. Note that Alg. 1 can only process successfully process parametric Statechars containing transitions of the form $s \xrightarrow{e} s'$ where s has no other outgoing transition. This is a limitation of the theoretical framework, but this does not limit the modelling of apps as sensing/actuating operations are almost always done exclusively with subsequent transitions handling decision making.



(a) A parametric statechart design for diabetes management



(b) Customising the diabetes plan with different numbers of *OR* and *AND* sensor aggregations

Figure 5: Additional m-Health app designs using ReHash

6. RESULTS

A tool-chain implementing the translation of parametric statecharts to static statecharts and then to Android code has been implemented in Java. The overall system, called ReHash, contains a a number of parsers to perform these translations[1].

To carry out a preliminary analysis of our approach, we modelled a number of m-Health using our formalism. These include a more details version of the fall assessment and detection plan shown in Fig. 3, a diabetes management plan shown in Fig. 5a, a weight management plan, and a medication reminder plan. Overall the parametric statecharts were similar in size to the plans shown in Fig. 3 and 4.

Fig. 5b shows the size of the static statechart obtained from Alg. 1 when a diabetes management app shown in Fig. 5a was processed with varying numbers of *OR* and *AND* operations based on the underlying smart-home configuration. Instead of supporting multiple sensors for temperature sensing, we used the *OR* case to log any of the previous values read by the temperature sensor (stored on a database) in the last few hours. Similarly, reading insulin levels can be based on taking a current reading and a specified number of previous readings stored on the database. Since these readings need to be averaged to compute the meta-variable *insulin*, we require all these readings to be read concurrently and then averaged. Fig. 5b shows that the size of the static statecharts being generated rises linearly with an increase in *OR* and *AND* operations in a given configuration.

Our tool consistently takes less than 500 milliseconds to convert a parametric statechart to a static statechart, and less than 2 seconds to generate the Android code. Hence, a change in a smart-home configuration causes an automatic re-translation which takes less than 2.5 seconds in total, excluding server and network delays. The size of the Android code including the XML and Java code created for each app was proportional to the size of the static statechart. Further experiments will be carried out over different kinds of application areas, such as home automation and industrial automation, to more comprehensively evaluate the performance of this solution.

Key future work includes formulating domain-specific languages that can convert apps designed by domain experts such as health-care professionals into parametric statecharts automatically, incorporating more complex sensor-fusion techniques, and more extensive benchmarking of the ReHash tool. Exploring the potential of our technique in designing successful m-Health apps will require an evaluation of how well these apps meet care requirements, and how easy it is for clinicians to design these apps. The current tool requires a few manual implementation-specific steps, such as creating a new update of the app once a design or configuration changes. Robust tool support is therefore required to automate the complete process.

[1]Software code for the project can be downloaded from https://git.io/vPjB0.

7. SUMMARY

We described the novel use of parametric statecharts for designing apps for smart-spaces in a flexible manner. These apps can be automatically customised to the configuration of a given IoT system which is a possibly dynamic set of available sensors and actuators. Any change in the configuration triggers and automatic code generation and deployment. An end-to-end compilation or re-compilation of plans takes at most 2.5 seconds for case studies considered in our research so far.

8. REFERENCES

[1] W. Cazzola, A. Ghoneim, and G. Saake. Software evolution through dynamic adaptation of its oo design. In *Objects, Agents, and Features*, pages 67–80. Springer, 2004.

[2] A. Chapko, B. Feodoroff, D. Werth, and P. Loos. A personalized and context-aware mobile assistance system for cardiovascular prevention and rehabilitation. *Lebensqualität im Wandel von Demografie und Technik*, 2013.

[3] R. Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65–99, 2009.

[4] G. Fortino, W. Russo, and E. Zimeo. A statecharts-based software development process for mobile agents. *Information and Software Technology*, 46(13):907–921, 2004.

[5] M.-P. Gagnon, P. Ngangue, J. Payne-Gagnon, and M. Desmartis. m-health adoption by healthcare professionals: a systematic review. *Journal of the*

American Medical Informatics Association, 23(1):212–220, 2016.

[6] T. Gao, D. Greenspan, M. Welsh, R. R. Juang, and A. Alm. Vital signs monitoring and patient tracking over a wireless network. In *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pages 102–105. IEEE, 2006.

[7] G. Gard and A. Larsson. Focus on motivation in the work rehabilitation planning process: a qualitative study from the employer's perspective. *Journal of Occupational Rehabilitation*, 13(3):159–167, 2003.

[8] C. V. Granger, B. B. Hamilton, R. A. Keith, M. Zielezny, and F. S. Sherwin. Advances in functional assessment for medical rehabilitation. *Topics in geriatric rehabilitation*, 1(3):59–74, 1986.

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

[10] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of uml statechart diagrams. In *Formal Methods for Open Object-Based Distributed Systems*, pages 331–347. Springer, 1999.

[11] N. Leveson, M. Heimdahl, H. Hildreth, J. Reese, and R. Ortega. Experiences using statecharts for a system requirements specification. In *Proceedings of the 6th international workshop on Software specification and design*, pages 31–41. IEEE Computer Society Press, 1991.

[12] Y. Masuda, M. Sekimoto, M. Nambu, Y. Higashi, T. Fujimoto, K. Chihara, and T. Tamura. An unconstrained monitoring system for home rehabilitation. *Engineering in Medicine and Biology Magazine, IEEE*, 24(4):43–47, 2005.

[13] A. Muelder. Yakindu. *Yakindu Statechart Modeling Tools*, 2011.

[14] S. Patel, H. Park, P. Bonato, L. Chan, and M. Rodgers. A review of wearable sensors and systems with application in rehabilitation. *Journal of neuroengineering and rehabilitation*, 9(1):1, 2012.

[15] G. Postolache, P. Silva GiraÌČo, and O. Postolache. Applying smartphone apps to drive greater patient engagement in personalized physiotherapy. In *Medical Measurements and Applications (MeMeA), 2014 IEEE International Symposium on*, pages 1–6. IEEE, 2014.

[16] C. Rougier, J. Meunier, A. St-Arnaud, and J. Rousseau. Fall detection from human shape and motion history using video surveillance. In *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on*, volume 2, pages 875–880. IEEE, 2007.

[17] S. Saeki, H. Ogata, T. Okubo, K. Takahashi, and T. Hoshuyama. Impact of factors indicating a poor prognosis on stroke rehabilitation effectiveness. *Clinical rehabilitation*, 7(2):99–104, 1993.

[18] P. R. Sama, Z. J. Eapen, K. P. Weinfurt, B. R. Shah, and K. A. Schulman. An evaluation of mobile health application tools. *JMIR mHealth and uHealth*, 2(2):e19, 2014.

[19] F. Sanfilippo and K. Pettersen. A sensor fusion wearable health-monitoring system with haptic feedback. In *Innovations in Information Technology (IIT), 2015 11th International Conference on*, pages 262–266. IEEE, 2015.

[20] B. Sobolev, D. Harel, C. Vasilakis, and A. Levy. Using the statecharts paradigm for simulation of patient flow in surgical care. *Health Care Management Science*, 11(1):79–86, 2008.

[21] N. K. Suryadevara and S. C. Mukhopadhyay. Wireless sensor network based home monitoring system for wellness determination of elderly. *IEEE Sensors Journal*, 12(6):1965–1972, 2012.

[22] M. Von der Beeck. A comparison of statecharts variants. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148. Springer, 1994.

[23] D. L. Walters, A. Sarela, A. Fairfull, K. Neighbour, C. Cowen, B. Stephens, T. Sellwood, B. Sellwood, M. Steer, M. Aust, et al. A mobile phone-based care model for outpatient cardiac rehabilitation: the care assessment platform (cap). *BMC Cardiovascular Disorders*, 10(1):5, 2010.

[24] I. Warren, T. Weerasinghe, R. Maddison, and Y. Wang. Odintelehealth: A mobile service platform for telehealth. *Procedia Computer Science*, 5:681–688, 2011.

[25] A. Weiss, T. Herman, N. Giladi, and J. M. Hausdorff. Objective assessment of fall risk in parkinson's disease using a body-fixed sensor worn for 3 days. *PloS one*, 9(5):e96675, 2014.

[26] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 314–323. IEEE, 2000.