A Guideline Representation Language for Pervasive Healthcare

Nick (Lik San) Fung

A GUIDELINE REPRESENTATION LANGUAGE FOR PERVASIVE HEALTHCARE

Lik San Nick Fung

A GUIDELINE REPRESENTATION LANGUAGE FOR PERVASIVE HEALTHCARE

DISSERTATION

to obtain
the degree of doctor at the Universiteit Twente,
on the authority of the rector magnificus,
prof. dr. ir. A. Veldkamp,
on account of the decision of the Doctorate Board
to be publicly defended
on Wednesday 23 June 2021 at 16.45 hours

by

Lik San Nick Fung

born on the 23rd of October, 1989
in Hong Kong, Hong Kong

This dissertation has been approved by:

Supervisors
prof. dr. ir. H.J. Hermens
prof. dr. ir. M.J. van Sinderen

ISBN: 978-90-365-5193-9
DOI: 10.3990/1.9789036551939

Graduation Committee:

Chair / secretary:	prof. dr. J.N. Kok
Supervisors:	prof. dr. ir. H.J. Hermens prof. dr. ir. M.J. van Sinderen
Committee Members:	prof. dr. ir. L.J.M. Nieuwenhuis prof. dr. ir. M. Aksit prof. dr. G. Guizzardi prof. dr. P. Maret prof. dr. M.M.R. Hutten

Abstract

Chronic diseases represent one of the main health challenges in the 21st century. Their increasing prevalence, coupled with factors such as an ageing population and a growing lack of healthcare resources, have necessitated a shift from the traditional, episodic and responsive model of healthcare to a chronic and proactive model that emphasises patient empowerment. As a result, to support this shift, pervasive healthcare systems have been researched and developed as early as the 2000s with the aim to provide "healthcare to anyone, at anytime, and anywhere". While these systems are traditionally focused on the continuous monitoring of patients, the continual development of mobile hardware technologies has enabled the emergence of intelligent systems that can support patients autonomously with minimal manual intervention from care providers.

At the same time, patient care in the traditional healthcare setting is increasingly supported by the use of clinical practice guidelines, which document the current best clinical practice as supported by the latest scientific evidence. These guidelines aim to improve and ensure the quality of patient care by facilitating adherence to proven best practice; to support their adoption, computer-interpretable, guideline representation languages have been developed to formalise clinical guidelines such that they can be executed automatically by guideline-based computer systems. In this way, support can be seamlessly provided to clinicians in making the best possible, evidence-based decisions for the patient.

This research aims to extend evidence-based healthcare beyond the traditional healthcare setting by bringing computerised clinical practice guidelines to the free-living setting to provide pervasive and guideline-based decision support to patients. In general, guideline languages capture the control flow between the different tasks in

a guideline and thereby assume a centralised controller for executing them. However, for pervasive healthcare, such centralised system architectures may not be the most appropriate since system components may require dynamic reconfiguration in response to factors such as changing patient requirements and unreliable communications environments. Therefore, the main contribution of this research is a new guideline language that focuses on the data flow in guidelines, whereby tasks are modelled as processes that execute in parallel. By parallelising and dynamically distributing guideline knowledge, each device that constitutes the patient's pervasive healthcare system can be adapted in real-time and provide decision support independently of each other, thereby avoiding a single point of failure.

The new guideline language is developed by using formal methods and by following a model-driven methodology. Thus as part of this research, a formal and generic data flow model of disease management in pervasive healthcare is created; it comprises four types of processes, namely Monitoring (M), Analysis (A), Decision (D) and Effectuation (E), and six types of data flowing between them, namely Measurement, Observation, Abstraction, Action Plan, Action Instruction and Control Instruction. This model is given a precise mathematical interpretation using axiomatic set theory, the result of which is divided into two complementary models. The first is a reference information model for representing the data flow, which comprises 32 set definitions, while the second is a guideline model for representing the MADE processes, which comprises 28 set definitions as well as 13 function signatures and 38 logical invariants to specify their behaviour.

From the reference information model and guideline model, the syntax and semantics of the new guideline language are derived for representing clinical guidelines. To support the verification and validation of the formalised guidelines, the syntax and semantics of an accompanying archetype language were also developed for specifying MADE archetypes (i.e. well-formedness constraints on MADE data items). Furthermore, a reference implementation for these two languages is developed which comprises a set of libraries implemented on top of Rosette. Since Rosette provides support for not only executing the languages but also verifying them using off-the-shelf constraint solvers, the reference implementation is formally verified to comply with the 38 logical invariants of the guideline model. More specifically,

these constraint solvers help ensure that no patient data will ever cause the reference implementation to violate its invariants during execution.

Validation of the MADE models and languages is conducted by formalising, verifying and testing a complete clinical guideline for gestational diabetes mellitus, i.e. diabetes that is first developed or first recognised during pregnancy. The guideline comprises 13 semi-formal workflows, such as for managing blood glucose levels and urinary ketone levels, and the result is a MADE network comprising 55 processes, specifically 0 Monitoring, 4 Analysis, 22 Decision and 29 Effectuation processes, all of which are connected together by the flow of 50 types of MADE data, specifically 0 Measurement, 8 Observation, 8 Abstraction, 14 Action Plan, 3 Action Plan and 17 Control Instruction archetypes. The 0 Monitoring processes and 0 Measurement archetypes were a result of the fact that all measurement tasks in the guideline were to be performed manually by the patient.

In the future, the clinical relevance of the MADE languages should be evaluated by developing and performing clinical studies on pervasive healthcare systems that implement the MADE languages. Usability of the languages should also be evaluated in collaboration with clinicians, patients and other stakeholders; improvements may include adding support for personalising guideline knowledge and for partial specifications of manual processes. A knowledge acquisition tool suite may also be developed from the reference implementation to provide support for formalising clinical guidelines, developing and executing test data as well as visualising the test and verification results.

Acknowledgements

It has been over 9 years since I embarked on this PhD project in 2012, and the whole experience has truly been life-changing for me. Indeed, without the support of many very wonderful people, I would not have progressed this far, so let me express my gratitude here to everyone that has supported me along the way. I will try to make sure everyone is mentioned, but please do forgive me if I've missed anyone.

Firstly, I would like to thank Prof. Dr. Ir. Hermie Hermens and Dr. Val Jones for being my supervisors since the start of my PhD and for giving me the opportunity to conduct research in pervasive healthcare as part of the MobiGuide project. To Dr. Val Jones, our frequent discussions have been very insightful for me, and I'm also grateful for your continued support even after your retirement. Many thanks also go to Prof. Dr. Ir. Marten van Sinderen, who agreed to become my supervisor after the retirement of Dr. Val Jones. While this all happened in the later stages of my PhD, your feedback on my work is still as valuable. To all my committee members, namely Prof. Dr. Ir. Nieuwenhuis, Prof. Dr. Ir. Aksit, Prof. Dr. Guizzardi, Prof. Dr. Maret and Prof. Dr. Hutten, thank you very much for taking the time to review and provide feedback on my thesis.

My research originated from the MobiGuide project, and I am glad to have been able to work with some amazing people in such a large, multidisciplinary team. These people include: Dr. Tom Broens, Daniel Knoppel, Dr. Erez Shalom, Dr. Gema García-Sáez, Dr. Iñaki Martínez-Sarriegui and Carlos Marcos, all of whom I've worked closely with to develop prototypes of the MobiGuide system. To Prof. Mor Peleg, Prof. Yuval Shahar, Prof. Silvana Quaglini and Prof. Elena Hernando, thank you for your general support, feedback and discussions.

I am also grateful to be part of the Biomedical Signals and Systems (BSS) group at the University of Twente, for introducing me to the Dutch culture amongst other things. To Sandra and Wies, the current and former secretary of BSS, thank you for helping me take care of many practical matters, and to Bart, Nekane, Jan-Willem and Pravin, thank you for being my office mates. It was nice working with you, Nekane, as fellow members of MobiGuide, and Bart, I had fun taking part in your photoshooting and filming projects. Thank you also for being my de facto personal trainer.

In between my PhD, I had the opportunity to conduct research at the Unversity of Toronto, and for that, I would like to thank Prof. Marsha Chechik for welcoming me to her research group as well as Dr. Sahar Kokaly for introducing me to research in automotive safety assurance. To Alessio, your programming expertise was very helpful, and to the rest of the group, your presentations and discussions were all very insightful and entertaining; indeed, my thesis would have been very different if I wasn't introduced to your work. I am also grateful for all the time we've spent together and for all the willing test subjects in my cooking ventures. These people include: Alicia, Yi, Federico, Soukayna, Aamod, Victor, Anthony, Mike, Nick, Vincent, Torin, Wei-Bo, Caroline, Alex, and Leo.

Thank you, Federico, for keeping me company throughout my time in Toronto. You were a great social chair, and I had fun working on assignments with you (again and again). To Johnny, thank you for all the get-togethers and outings, and to Victor, thank you for establishing a standard time for the group. It's a pity that I couldn't do much cycling and skiing with you. To Mike, I enjoyed working with you on assurance cases, and needless to say, you were a great flatmate and gym buddy. To Aamod, thank you for sharing with us your expertise in tabletop games.

Finally, special thanks go to my family, namely my parents for raising me up and my sister for being my best and the worst sibling. It's comforting to know that there's always a place I can return to in Hong Kong, and whenever I do, I'm also glad that Paul, Martin and Calvin are there to spend time with me. To Georg, thank you for regularly encouraging me and for being my de facto tour guide during my PhD.

Table of Contents

List of Figures

List of Tables

List of Terms

AF See Atrial Fibrillation.

Analysis In the context of MADE, the diagnostic process of making high-level abstractions from low-level observations.

Atrial Fibrillation Abnormal heart rhythm originating from the patient's atria.

CIM See Computation Independent Model.

Clinical Practice Guideline A collection of statements (including recommendations) that are intended to optimize patient care and are informed by a systematic review of evidence and an assessment of the potential risks and benefits of possible alternatives.

Computation Independent Model A model of the application domain without reference to any system-related details.

CPG See Clinical Practice Guideline.

Decision In the context of MADE, the therapeutic decision-making process of deciding on the appropriate plan of action given the current state of the environment (which encompasses both the patient and their surroundings).

DIM See Domain Information Model.

Domain Information Model A specialisation of a RIM for a specific application.

Effectuation In the context of MADE, the process of performing the decided action plan, with the intention of bringing about a change in the patient's state.

Formal Methods Mathematically-based techniques for specifying, developing and verifying software systems.

GDM See Gestational Diabetes Mellitus.

Gestational Diabetes Mellitus The manifestation of diabetes during pregnancy.

GLM See Guideline Model.

Guideline Model For this thesis, a generic model of the tasks specified in clinical guidelines.

Guideline-based System A computer system designed to execute formalised guidelines (i.e. guidelines that are represented in a computer-interpretable form).

MADE See Monitoring, Analysis, Decision and Effectuation.

MDA See Model-driven Architecture.

Model-driven Architecture A model-driven approach to software design, development and implementation developed by the Object Management Group (OMG).

Model-driven Engineering A software development methodology that emphasises the use of domain models in the development of software systems.

Monitoring In the context of MADE, the process of making observations about the environment, including but not necessarily limited to the patient.

NCD Noncommunicable disease. Also known as chronic disease.

Pervasive Healthcare System A computer system designed to provide healthcare to patients at anytime and anywhere.

PIM See Platform Independent Model.

Platform Specific Model A model which contains all the relevant details for implementing the software system on a specific platform.

Platform Independent Model A model of the software system without reference to any implementation specific details.

proxy In the context of MADE, data items or processes that output data items that require manual confirmation before they can be processed further by other MADE processes. In other words, proxy data items are not to be processed automatically by MADE processes.

PSM See Platform Specific Model.

Reference Information Model A generic model of data for a problem domain independent of any specific applications. For this thesis, the RIM represents all clinical data that may be relevant for guideline-based pervasive healthcare.

RIM See Reference Information Model.

List of Publications

Below is the list of first-author and co-author publications to support this thesis.

Journal Articles

Fung, N. L. S., Jones, V. M., Widya, I., Broens, T. H. F., Larburu, N., Bults, R. G. A., Shalom, E., and Hermens, H. J. (2016). The Conceptual MADE Framework for Pervasive and Knowledge-Based Decision Support in Telemedicine. *International Journal of Knowledge and Systems Science*, 7(1):25–39.

Fung, N. L. S., Jones, V. M., and Hermens, H. J. (2017). The MADE reference information model for interoperable pervasive telemedicine systems. *Methods of Information in Medicine*, 56(2):180–186.

Peleg, M., Shahar, Y., Quaglini, S., Broens, T., Budasu, R., **Fung, N.**, Fux, A., García-Sáez, G., Goldstein, A., González-Ferrer, A., Hermens, H., Hernando, M. E., Jones, V., Klebanov, G., Klimov, D., Knoppel, D., Larburu, N., Marcos, C., Martínez-Sarriegui, I., Napolitano, C., Àngels Pallàs, Palomares, A., Parimbelli, E., Pons, B., Rigla, M., Sacchi, L., Shalom, E., Soffer, P., and van Schooten, B. (2017). Assessment of a personalized and distributed patient guidance system. *International Journal of Medical Informatics*, 101:108–130.

Conference and Workshop Papers

González-Ferrer, A., Peleg, M., Parimbelli, E., Shalom, E., Marcos, C., Klebanov, G., Martinez-Sarriegui, I., **Fung, N. L. S.**, and Broens, T. (2014). Use of the

virtual medical record data model for communication among components of a distributed decision-support system. In *2014 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*, pages 526–530.

Fung, N. L. S., Widya, I., Broens, T., Larburu, N., Bults, R., Shalom, E., Jones, V., and Hermens, H. (2014). Application of a Conceptual Framework for the Modelling and Execution of Clinical Guidelines as Networks of Concurrent Processes. *Procedia Computer Science*, 35:671–680.

Fung, N. L. S., Jones, V. M., Bults, R. G. A., and Hermens, H. J. (2014). Guideline-based Decision Support for the Mobile Patient Incorporating Data Streams from a Body Sensor Network. In *4th International Conference on Wireless Mobile Communication and Healthcare*.

Shalom, E., Shahar, Y., Goldstein, A., Ariel, E., Quaglini, S., Sacchi, L., **Fung, N.**, Jones, V., Broens, T., García-Sáez, G., and Hernando, E. (2015). Enhancing Guideline-Based Decision Support with Distributed Computation Through Local Mobile Application. In Fournier, F. and Mendling, J., editors, *Business Process Management Workshops*, pages 53–58, Cham. Springer International Publishing.

Shalom, E., Shahar, Y., Goldstein, A., Ariel, E., Sheinberger, M., **Fung, N.**, Jones, V., and van Schooten, B. (2015b). Implementation of a Distributed Guideline-Based Decision Support Model Within a Patient-Guidance Framework. In Riaño, D., Lenz, R., Miksch, S., Peleg, M., Reichert, M., and ten Teije, A., editors, *Knowledge Representation for Health Care*, pages 111–125. Springer International Publishing.

Larburu, N., van Schooten, B., Shalom, E., **Fung, N.**, van Sinderen, M., Hermens, H., and Jones, V. (2015b). A Quality-of-Data Aware Mobile Decision Support System for Patients with Chronic Illnesses. In Riaño, D., Lenz, R., Miksch, S., Peleg, M., Reichert, M., and ten Teije, A., editors, *Knowledge Representation for Health Care*, pages 126–139, Cham. Springer International Publishing.

Fung, N. L. S., van Sinderen, M. J., Jones, V. M., and Hermens, H. J. (2020). A verified, executable formalism for resilient and pervasive guideline-based decision support for patients. In Michalowski, M. and Moskovitch, R., ed-

itors, *Artificial Intelligence in Medicine*, pages 427–439, Cham. Springer International Publishing.

Chapter 1

Introduction

1.1 Motivation

Noncommunicable diseases (NCDs), also known as chronic diseases, are a major cause of death worldwide (around 70 % in 2016) and are recognized by the World Health Organization as one of the main health challenges in the 21st century [91]. Unlike acute diseases, NCDs require consistent, long-term management, necessitating a shift from the traditional, episodic and responsive model of healthcare to a chronic care model that emphasises patient empowerment and proactive clinical practice [64, 87]. However, providing chronic care is not without challenges, especially with an aging population and an increasing lack of healthcare resources [3]. As a result, as early as the 2000s, research has been conducted on the use of computer systems to support pervasive healthcare, which aim to provide "healthcare to anyone, at anytime, and anywhere" [86].

Traditionally, pervasive healthcare systems focused on the continuous monitoring of patients, but with increasing development of mobile hardware technologies, there is growing research on intelligent systems that can automate data processing and analysis [2]. This is necessitated by the volumes of data that can be collected by the systems about each individual patient, all of which can easily lead to information overload and hamper clinical decision making as evidenced in the hospital setting [17]. Therefore, the ideal pervasive healthcare system should not only monitor patients during their daily lives but also provide support to them automatically without any

manual intervention, thereby allowing caregivers to focus on the most important aspects of patient care.

However, the use of such autonomous pervasive healthcare system does not preclude the need to provide the best possible care to patients. In the traditional hospital setting, the quality of patient care is assured via the use of clinical practice guidelines (CPGs), which are defined as "statements that include recommendations intended to optimize patient care that are informed by a systematic review of evidence and an assessment of the benefits and harms of alternative care options" [31]. While these guidelines are generally written in narrative English, guideline-based computer systems have been developed to execute formalised versions of guidelines to seamlessly support clinicians in making the best possible, evidence-based decisions for the patient.

Therefore, to provide evidence-based support to patients in their daily lives, this thesis aims to bridge the gap between pervasive healthcare systems for patients and guideline-based systems for clinicians. In Sec. 1.2, the main features of these two types of systems are presented, which leads to the research objectives and questions detailed in Sec. 1.3. Sec. 1.4 and 1.5 outline, respectively, the context and scope within which this research is conducted, while in Sec. 1.6, the research approach and outputs are described. Finally, an outline of the remainder of the thesis is presented in Sec. 1.7.

1.2 System Feature Analysis

1.2.1 Pervasive Healthcare Systems

As mentioned in Sec. 1.1, pervasive healthcare systems are designed to support patients at any time and anywhere, which implies that they are generally data-driven and are required to process data streams in real time. Specialised sensors (e.g. blood glucose and inertial sensors) continually collect data about the patient and their environment, all of which are then processed to determine the appropriate support to provide to the patient [60, 30]. This support can include immediate alerts when abnormalities are detected (e.g. for fall detection [28]) as well as automatic activation of the appropriate actuators (e.g. for insulin management [95]).

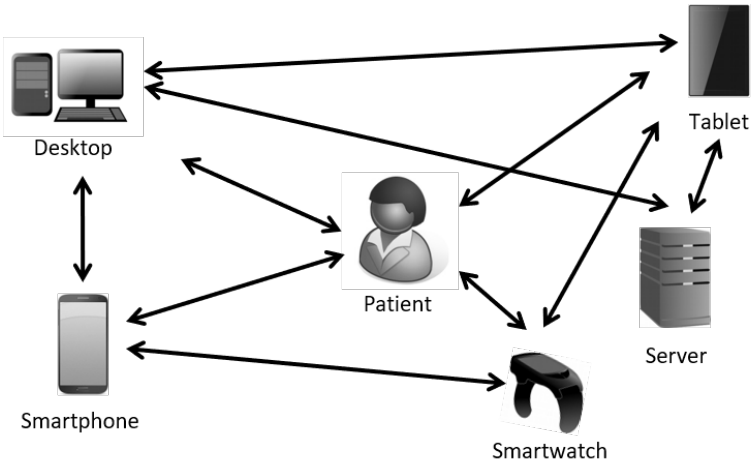

Fig. 1.1 Example composition of a generic pervasive healthcare system.

Given their need for sensors, data processors as well as actuators, pervasive healthcare systems are typically distributed in nature. In fact, while data processing often occurs in the back-end, where advances in high performance and cloud computing allow for highly expensive operations (e.g. deep learning), some data processing tasks may be best distributed to front-end components (e.g. smartphones or personal computers). Communications networks are not always reliable, but a reliable pervasive healthcare system should always provide the appropriate support regardless. Thus in the most generic case, a pervasive healthcare system may comprise a network of devices (such as shown in Fig. 1.1) and adopt a "holonic" multi-agent system architecture, with each device capable of providing support independently of each other but sharing data as well as functionality [49] to achieve overall system resilience against technical disruptions.

1.2.2 Guideline-based Systems

Guideline-based systems comprise three main components as shown in Fig. 1.2 [33]:

- A knowledge base containing the formalised clinical guidelines.
- A patient database containing all the necessary patient data.
- A guideline execution engine to execute the stored clinical guidelines.

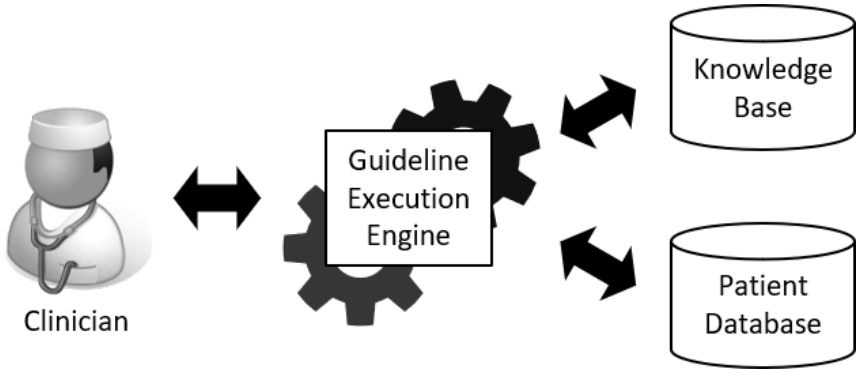

Fig. 1.2 Typical components of a guideline-based system.

During runtime, the clinician invokes the guideline execution engine and selects the appropriate guideline to execute and the target patient to execute the guideline on. The engine then begins executing the guideline, querying the database whenever necessary to retrieve the relevant patient data. The execution engine may also send queries to the clinician if the necessary data is unavailable or provide recommendations if dictated by the guideline, the results for which may then be fed back into the engine to continue the guideline execution. Inputs to the system during runtime may also be stored in the knowledge base (if they relate to clinical guidelines) or in the database (if they relate to patients).

As a sub-type of knowledge-based systems, a key feature of guideline-based systems is the explicit representation of knowledge [8]. For example, one of the earliest knowledge-based systems for clinical decision support is MYCIN [72], which reasons with clinical knowledge represented in the form of production rules. Thus while pervasive healthcare systems may be designed in consultation with clinical experts, they are generally not guideline-based as the clinical knowledge is embedded within the source code of the system. By separating the knowledge from the system implementation, guideline-based systems can be easily adapted to execute different guidelines, provided the guidelines can all be represented using the chosen computer-interpretable, guideline representation language. This is also implies a common representation for all patient data as different guidelines may require different data.

1.3 Research Questions

Guideline-based pervasive healthcare systems offer the advantages of being explicitly evidence-based and application independent; an explicit representation of knowledge can help assure that the system is always providing the best evidence-based support to patients, and by formalizing and executing different guidelines, the same system can be adapted to support patients with different conditions. However, unlike traditional guideline-based systems that can be deployed on a fixed back-end infrastructure, guideline-based pervasive healthcare systems should also be highly adaptable to changing physical configurations. This ensures that the provided support remains unaffected by factors such as unreliable communications networks and changing patient needs, which leads to the main research question (RQ) of this thesis:

Main RQ. How can pervasive and knowledge-based support be provided to patients?

One key challenge in developing knowledge-based systems, including guideline-based systems, is designing the appropriate computer-interpretable language for representing the desired knowledge. Different knowledge representation languages require different reasoning procedures, and in general, languages with higher expressive power are also more computationally expensive [8]. Therefore, a careful balance must be achieved to ensure that the language is appropriate for pervasive healthcare systems and is sufficiently expressive for clinical guidelines without being unnecessarily inefficient. As a result, the following sub-question can be derived from the main research question:

RQ 1. What is an appropriate knowledge representation language for formalising clinical guidelines for pervasive healthcare systems?

Apart from its expressiveness and computational complexity, the design of a guideline representation language is also influenced by the chosen representation of patient data, which is highly heterogeneous in nature. For example, patient data not only includes raw numerical measurements but also abstract clinical diagnoses and complex drug regimens. This raises the following question:

RQ 2. What is an appropriate representation for patient data in the context of pervasive healthcare?

Given their differing syntax and semantics, different guideline representation languages require different guideline execution engines to interpret them. Therefore, as part of this thesis, the following research question will also be addressed:

RQ 3. What is an appropriate design and implementation of the guideline execution engine for guideline-based pervasive healthcare systems?

Finally, formalizing clinical guidelines is a non-trivial task; it may require a collaboration between multiple parties and involve multiple activities, from selecting the appropriate guideline for the target application to establishing a consensus on the guideline semantics and evaluating the formalised guideline [71]. While this thesis does not aim to develop a knowledge acquisition tool, it is hypothesised that the guideline execution engine, which can reason with clinical knowledge, can be adapted to support the verification and validation of formalised guidelines. Thus the final research question of this thesis is as follows:

RQ 4. What extensions can be incorporated into the guideline execution engine (and associated language) to support the verification and validation of formalised clinical guidelines?

1.4 Context: The MobiGuide Project

The research presented in this thesis forms part of and builds upon the MobiGuide project (https://cordis.europa.eu/project/id/287811), which is conducted under the European Seventh Framework Program (FP7). Similar to this research, the MobiGuide project aims to develop a guideline-based decision support system for chronic patients during their daily lives. However, in this MobiGuide project, a centralised system architecture is adopted that comprises two main computing components [56]:

- A smartphone that remains by the patient, processing patient data locally to ensure that support is provided whenever and wherever necessary.
- A back-end server that resides at the hospital, performing any necessary but expensive tasks and providing decision support to clinicians. This back-end server is also responsible for “projecting” the appropriate portions of a clinical guideline to the smartphone component for local execution [70].

This thesis extends the MobiGuide project by considering the general case wherein decision support is provided by an "n-ary" pervasive healthcare system, i.e. by an arbitrary number of devices. Furthermore, this thesis assumes a "holonic" multi-agent architecture which, as described in Sec. 1.2.1, can offer increased robustness compared to a centralised architecture. In this respect, the MobiGuide system can be seen as a specific instance of an "n-ary" pervasive healthcare system, with n equal to 2, and it therefore serves as a useful frame of reference for this thesis. In fact, the research presented in this thesis is validated using a clinical guideline that has been developed as part of the MobiGuide project and used to validate the clinical relevance of the MobiGuide system. This guideline is targeted at patients with gestational diabetes mellitus (GDM), i.e. patients who exhibit diabetes during pregnancy.

1.5 Research Scope

As implied by the research questions presented in Sec. 1.3, the main aim of this research is to enable the formalisation and execution of clinical guidelines in the context of pervasive healthcare. However, it should be noted that to truly realise a guideline-based pervasive healthcare system, many other considerations must also be taken into account which are outside the scope of this thesis. For example, given that pervasive healthcare systems may receive data from and output data to multiple locations, including sensors and actuators on the patient as well as electronic health records (EHRs) in hospitals, one technical concern is the design and implementation of an appropriate mechanism to accumulate all input and output data.

To support such integration of medical data into traditional guideline-based systems, "mediator" components have been proposed, such as KDOM by Peleg, Keren and Denekamp in 2008 [55] and IDAN by Boaz and Shahar in 2005 [6]. KDOM is a framework that maps abstractions found in clinical guidelines to specific data items stored in EHRs [55], and similarly, IDAN provides a generic interface for healthcare information systems to access and query heterogeneous clinical data sources [6]. Such components may also be integrated into guideline-based pervasive healthcare systems, but these technical considerations are outside the scope of this thesis, which focuses on the clinical aspects of representing clinical knowledge and data.

Indeed, while this research addresses the design and implementation of a guideline execution engine for pervasive healthcare, its focus is on providing a reference implementation to demonstrate the semantics of the developed guideline representation language. As a result, technical performance issues, such as the potential need to process large data streams in real-time, are ignored. Furthermore, this research does not address the specific mechanisms with which a pervasive healthcare system can distribute its functionality across its devices. As mentioned in Sec. 1.2.1, such a mechanism is necessary to mitigate issues caused by unreliable communications networks and other technical disruptions. However, the focus of this thesis is not on the actual distribution of functionality per se, but on enabling such a distribution via the formalisation of clinical guideline knowledge.

At this point, it should also be mentioned that as early as the 2000s, research has been conducted on the formalisation of clinical guidelines for verification purposes. For example, in 2006, Giordano et al. described how model checking techniques for linear temporal logic can be used to verify certain properties of clinical guidelines [23], while ten Teije et al. demonstrated the use of the KIV theorem prover and interval temporal logic for a similar purpose [79]. More recently, Wilk et al. and Michalowski et al. presented in 2013 and 2020 respectively two different approaches to detect and mitigate conflicts in the concurrent application of multiple guidelines; the approach by Wilk et al. relies on constraint logic programming [88] while that by Michaloski et al. uses first-order logic [47]. While this thesis also touches upon this issue of verifying clinical guidelines, the focus is on executable representations of clinical guidelines as opposed to representations that can be reasoned with. Thus the use of such formal systems are considered outside the scope of this research.

Likewise, as implied in Sec. 1.3, issues relating to the usability and acceptability of guideline-based systems are not investigated. For example, Goud, Hasman and Peek in 2008 presented a guideline-based system (CARDSS) which contains, amongst other usability features, explanation facilities to increase the acceptance of any output recommendations [27]. Furthermore, in 2010, Hatsek et al. presented the GESHER system to facilitate the formalisation and maintenance of clinical guidelines [29]. Although such considerations are also critical to the adoption of any guideline-based system, they are outside the scope of this thesis.

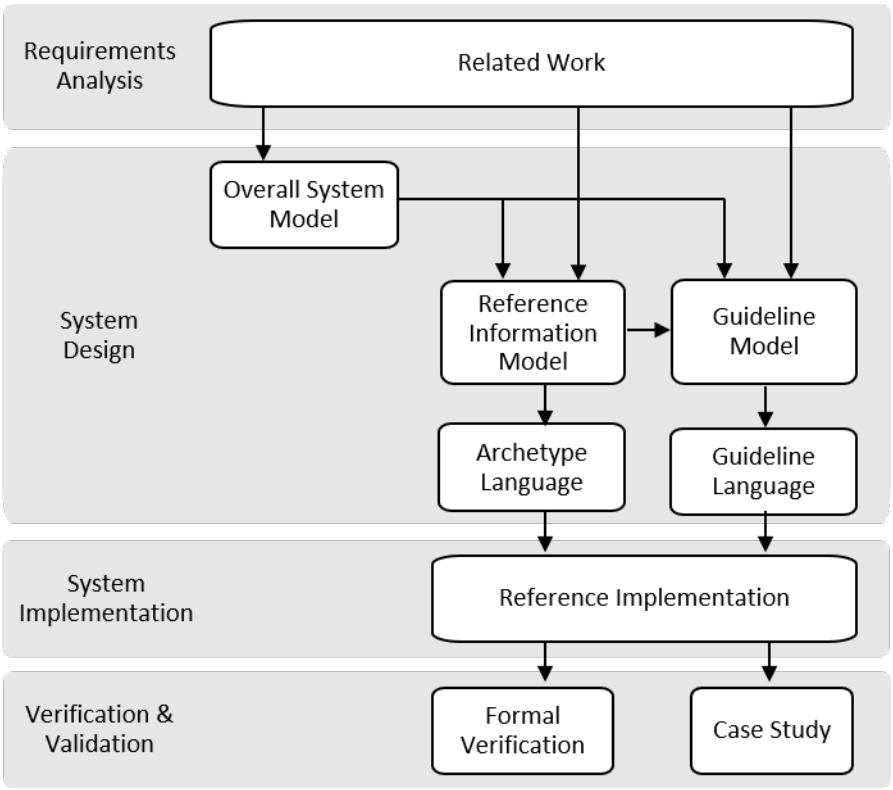

Fig. 1.3 The main research outputs of this thesis, mapped to the corresponding software development phase. $A \rightarrow B$ indicates that B depends directly on A.

1.6 Research Approach

The development of software systems typically involve four main phases [75]: requirements analysis, system design, system implementation, and finally verification and validation. While these phases may occur iteratively and concurrently (as is the case for agile development methodologies), this PhD thesis adopts a linear, waterfall approach to address the research questions presented in Sec. 1.3. The resulting research outputs are summarised in Fig. 1.3.

Firstly, for requirements analysis, related work is studied to identify the key features and limitations of existing guideline-based systems and guideline representation languages, the results of which are then used in the system design phase. This thesis

adopts a model-driven development methodology, thus the outputs of the design phase include various models on which the proposed guideline language is based, namely:

- The overall system model, which captures the key features of pervasive healthcare systems and presents the main design decisions underlying the proposed guideline language.
- The reference information model (RIM), which extends the overall system model and specifies the types of patient data that may be required by pervasive healthcare systems.
- The guideline model (GLM), which is based on the overall system model and reference information model and captures the components that may constitute a clinical guideline.

To ensure that the requirements are satisfied, formal methods are adopted such that the three models are given a precise mathematical interpretation. In particular, all models developed in this thesis are specified formally using axiomatic set theory, the notation for which is summarised in Table 1.1. As implied in the table, a set is treated in this thesis as largely synonymous to a data type in the sense used in programming languages. Thus, for example, the statement "ELEM is an element of set T" can be interpreted as being equivalent to "ELEM is an instance of type T".

Having formalised the models, the next step as illustrated in Fig. 1.3 is to derive from those models the syntax and semantics of the proposed guideline representation language, including constructs for verifying and validating formalised guidelines. In fact, an archetype language is also derived from the reference information model, which enables the specification of well-formedness constraints on patient data and thereby provides additional support for verifying and validating clinical guidelines. For example, an archetype for body temperature might indicate that it is a numerical quantity measured in $^{\circ}C$ or $^{\circ}F$, thus any guideline that violates this constraint would automatically be considered incorrect.

While this thesis does not aim to produce a commercial knowledge-based pervasive healthcare system, an implementation of the guideline and archetype languages is provided to demonstrate that they are indeed implementable and to serve as a gold

Table 1.1 Summary of the mathematical notation adopted for modelling pervasive healthcare.

Notation	**Description**
TypeName	The name of a set, i.e. a collection of elements.
$Type_1 = Type_2$	$Type_1$ is the same (i.e. contain the same elements) as $Type_2$.
$T = \{\text{ELEM}_1, \text{ELEM}_2\}$	Set T contains two elements, ELEM_1 and ELEM_2.
$T = T_1 \cup T_2$	Set T is the union of (i.e. comprises all the elements of) sets T_1 and T_2.
$T = T_1 \cap T_2$	Set T is the intersection of (i.e. comprises only the elements shared in) sets T_1 and T_2.
$T = T_1 \times T_2 \times \cdots \times T_N$	Each element of T is a tuple comprising N components; the first is an element of $T1$, the second T_2, etc.
$T_1 \subset T_2$	T_1 is a proper subset of T_2. I.e. elements in T_1 are also in T_2 but not vice versa.
$T = \mathscr{P}(T_1)$	Elements of T are sets whose elements are of T_1.
$\pi_n(t)$	The nth component of element t.
$\text{ELEM} \in T$	ELEM is an element of T.

standard for future, possibly more efficient, implementations. The reference implementation is developed as set of libraries on top of Rosette, which provides support for not only executing the implementation but also analysing it using off-the-shelf constraint solvers [85]. Thus, the reference implementation is verified formally using Rosette against the properties of the original models.

Finally, the reference implementation, together with the guideline and archetype languages, are validated using a case study, specifically the gestational diabetes guideline from MobiGuide. As shown in Fig. 1.4, the clinical guideline is first formalised using the guideline and archetype languages, the result of which is a Rosette program. This program utilises the libraries provided by the reference implementation and can be executed using Rosette's interpreter. At this point, patient data can be provided to the program to generate decision support, but for this research, various Rosette commands are used instead to verify the formalised guideline and thereby validate the research outputs of this thesis.

1.7 Thesis Outline

Table 1.2 summarises the structure of this thesis. Except for related work, which is distributed across the thesis as appropriate, the chapters of this thesis largely correspond to the research outputs presented in Fig. 1.3. More specifically, the thesis starts in Ch. 2 with the necessary background on the model-driven development methodology and formal methods that underpin this research, followed by the details of the overall system model in Ch. 3. The next two chapters all relate to the representation of data: Ch. 4 on the reference information model and Ch. 5 on the archetype language (including its verified reference implementation). Similarly, Ch. 6 and 7 presents, respectively, the guideline model and the guideline language (including its verified reference implementation). Finally, the evaluation is presented; Ch. 8 focuses on the case study and Ch. 9 on the conclusions.

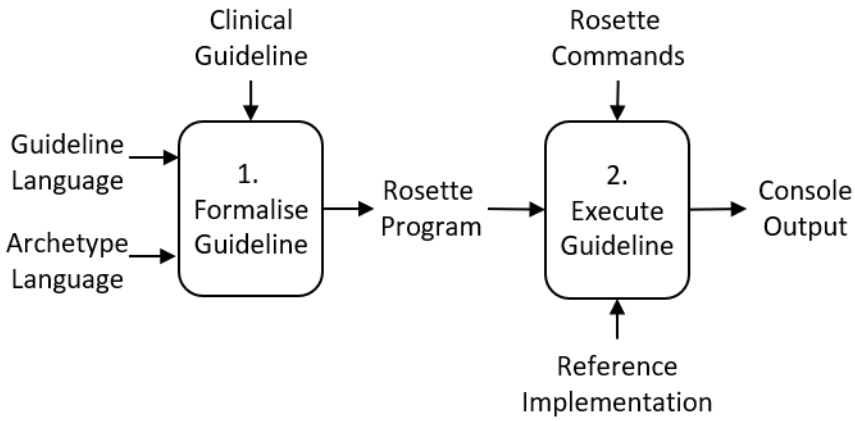

Fig. 1.4 The process followed for validating the reference implementation as well as the guideline and archetype languages.

Table 1.2 Contents of the thesis chapters and their relation to the research questions.

Ch.	Contents	Related RQs
1	Introduction	–
2	Background	–
3	Overall System Model	1, 2, 3
4	Reference Information Model	2, 3
5	Archetype Language & Implementation	3, 4
6	Guideline Model	1, 3
7	Guideline Language & Implementation	1, 4
8	Case Study	1, 2, 3, 4
9	Conclusions	–

Chapter 2

Methodology

2.1 Introduction

As described in Sec. 1.6, a model-driven methodology is adopted in conjunction with formal methods as part of the overall approach to address the research questions of this thesis. Therefore, in this chapter, the relevant background on these methods are presented; Sec. 2.2 focuses on model-driven engineering while Sec. 2.3 explores the different formal methods that were considered for this research.

2.2 Model-Driven Engineering

Model-driven engineering is a software development methodology that emphasises the use of domain models in the development of software systems [10]. Generally defined as simplified representations of the problem domain, different domain models can focus on different aspects of the target system (e.g. static components or dynamic behaviour) and can be at different levels of abstraction (from the overall architecture to implementation-specific details). In particular, the model-driven architecture (MDA) framework developed by the Object Management Group (OMG) distinguishes between three types of models [10, 40]:

- Computation independent model (CIM), which represents the application domain and captures the domain requirements without reference to any system-related details.
- Platform-independent model (PIM), which captures the functionality of the software system without reference to any implementation specific details.
- Platform-specific model (PSM), which specifies all the relevant details for implementing the software system on a specific platform (e.g. programming language and operating system).

The MDA framework is accompanied by various modelling standards and specifications, such as the Unified Modeling Language (UML) [54] and the MOF Model to Text Transformation Language (MOFM2T) [53], which not only allow these models to be created but also support automatic transformations between them. Indeed, much of these standards, especially UML, have been implemented in software tools to assist with the development of software systems; a notable example is the Eclipse Modeling Framework provided by the Eclipse IDE (https://www.eclipse.org/modeling/emf/).

While it is not the purpose of this PhD to be completely compliant with the MDA framework, the framework is loosely followed to ensure that the developed languages, and all related research outputs, are appropriate for providing guideline-based pervasive healthcare. Thus, for this PhD:

- The CIM corresponds to a conceptual model of the disease management process for patients with chronic diseases (which is presented in Ch. 3).
- The PIM specialises the CIM for guideline-based pervasive healthcare systems and is an aggregate of the overall system model (presented in Ch. 3), reference information model (Ch. 4) and guideline model (Ch. 6) that are developed as part of this thesis.
- The PSM specialises the PIM for a specific implementation and therefore corresponds to the reference implementation of the developed knowledge representation language (which is presented in Ch. 5 and 7).

2.3 Formal Methods

Formal methods are mathematically-based techniques for specifying, developing and verifying systems. For this thesis, the CIM and PIM are specified formally such that they are given a precise mathematical interpretation, which also opens up the possibility to formally verify that the reference implementation (i.e. the PSM) satisfies all the requirements, assumptions and properties specified in the PIM. Axiomatic set theory is used to specify the CIM and PIM due to its simplicity, general applicability and amenability to mathematical analysis, while four systems for developing and verifying the reference implementation were considered: Alloy [35], USE (UML-based Specification Environment) [24], VDM (Vienna Development Method) [19] and Rosette [85]. These four candidates were selected after consultation with experts in model-driven engineering and formal methods; as detailed in the following subsections, these systems all support the specification and verification of arbitrary models. However, Rosette is finally adopted since it is executable and can automate verification.

2.3.1 Alloy

Alloy (https://alloytools.org/) is based on relational logic, such that all aspects of the system are represented as relations (i.e. where relations are represented as sets of tuples) and operations over those relations [35]. To verify that the system model satisfies certain properties, assertions can be added to the specification which are checked automatically by Alloy using bounded model checking. In other words, for each assertion, Alloy searches for a counterexample within a bounded size to demonstrate that the assertion is invalid; if none is found, then either the assertion is valid or the bound should be relaxed to search for larger counterexamples.

While Alloy provides facilities for automated verification of system properties, it provides minimal support for defining complex functions to manipulate elements in the relations. In fact, it is not possible for the user to instantiate relations with concrete elements, which would be necessary to formalize and execute the GDM (gestational diabetes mellitus) guideline in the case study. As a result, Alloy is deemed inappropriate for the requirements of this research.

2.3.2 USE

As its name suggests, models are specified in USE (available at http://www.db.informatik.uni-bremen.de/projects/USE-2.3.1/) using UML [24], and unlike Alloy, USE allows the user to create specific instances of the specified models and perform operations on them. Furthermore, expressions in OCL (Object Constraint Language) can be incorporated into the models to specify invariants over them as well as to specify the pre- and post-conditions of the model operators. Every time a model is instantiated or operated on, USE can check whether any of the constraints are violated.

However, in USE, limited support is provided for formal verification; constraints are checked in relation to a specific instance of the model, thus it relies heavily on manual testing unlike Alloy, which can search for counterexamples automatically. For this research, this means that the reference implementation can only be verified in USE by formalising and testing different example guidelines, which may be impractical given their potential complexity.

2.3.3 VDM

Similar to USE, specifications in VDM [19] can be executable, allowing the user to prototype, explore and manipulate specific instances of models. Invariants, pre- and post-conditions can also be added to the specifications and checked during execution. However, tool support is also provided, specifically the Overture tool (http://overturetool.org/) for performing combinatorial testing as well as proof obligation generation. Combinatorial testing enables the user to automatically generate and execute large of number of test cases by iterating through combinations of sets of user-defined values, while proof obligations specify the conditions that must be proven to ensure that the specification, when executed, will not run into any errors.

While VDM provides more support than USE for verifying model instances, combinatorial testing and proof obligations may not be the most appropriate for this research. Since the models for this thesis should apply to all applications of guideline-based pervasive healthcare systems, the models are hypothesised to be very complex. Thus even with tool support, combinatorial testing may prove intractable and unable to cover the most significant edge cases. Furthermore, it is generally

accepted that formal proofs of correctness are too expensive to perform, thus proving correctness, while useful, is also considered outside the scope of this research.

2.3.4 Rosette

Rosette (https://emina.github.io/rosette/) is a language and system that extends the Racket programming language with constructs for program verification, debugging, synthesis as well as "angelic execution" [85]. Thus similar to USE and VDM, specifications implemented in Rosette are also executable, and concrete instances of the specifications can be checked during run-time against its invariants and pre- and post-conditions, all of which can be specified as assertions. However, one main advantage of Rosette over VDM and USE is that like Alloy, Rosette can automatically search for counter-examples that violate the specified assertions, thus not requiring the user to manually create test cases to checks for errors.

To achieve this, Rosette compiles the specifications into logical constraints that can be solved using off-the-shelf solvers (e.g. Z3 [18]). However, the main limitation of this approach is that these constraints are not always possible to solve and that it cannot be known in advance whether they are solvable or not. In other words, for this PhD, it is possible (and indeed likely) that the PIM may be too complex for Rosette to verify the reference implementation against. However, it is believed that Rosette can nevertheless provide useful insights for verifying the reference implementation against the PIM. Furthermore, testing is still an option since Rosette is executable, thus making Rosette the most appropriate out of the four candidate systems.

Chapter 3

The Architecture Model

3.1 Introduction

One of the key challenges in designing knowledge-based systems is choosing the appropriate representation of knowledge. As early as the 1980s, it is recognised that knowledge representation languages must achieve a fine balance between expressiveness and efficacy [90]; the chosen representation must be adequately expressive to capture the required concepts, but at the same time, it should not be unnecessarily complex. This ensures that represented knowledge can be maintained and reasoned with efficiently.

Therefore, to arrive at an appropriate knowledge representation for this thesis, a model-driven approach is employed in which the general process of managing a patient's condition is modelled and formalized. This chapter in particular focuses on the architecture models, which capture the "set of principal design decisions" about a system [78]. In the context of this thesis, these models correspond to the computation independent model (CIM) and the platform independent model (PIM) of the overall system, since they provide the foundation on which the four research questions are answered. The RIM and GLM, although part of the PIM, relate to detailed design decisions and are presented in subsequent chapters.

In Sec. 3.2, related models for representing clinical guidelines and disease management are presented, followed in Sec. 3.3 by a description and formal specification of a generic data flow model on which the rest of the thesis is based. This model

is independent of any domain, thus it is specialised into a CIM and overall PIM for pervasive healthcare, the complete details of which are presented in Sec. 3.4 and 3.5 respectively. The validation of the CIM and PIM using a complete clinical guideline is reserved for Ch. 8, but in Sec. 3.6, an application of the PIM is demonstrated, using a small portion of the GDM clinical practice guideline as an example. Finally, the models are discussed in Sec. 3.7.

3.2 Related Work

3.2.1 Control Flow Representations of Clinical Guidelines

Automated guideline-based systems have long been developed to provide decision support to clinicians [33] by executing formalised versions of clinical guidelines. Depending on the system's purpose, different formalisms have been adopted, which can be divided into the following three categories [58]:

- Document models that encapsulate the properties of guideline documents (e.g. their target audiences).
- Decision trees and probabilistic models that capture the algorithmic knowledge in guidelines.
- Task-network models that represent clinical guidelines as networks of tasks and may subsume the other models.

Of relevance to this thesis are the task-network models as they enable clinical guidelines to be formalised such that the complete disease management process can be automated. The main exemplars of task-network models include GLIF3 [7], ASBRU [68], GLARE [80], and PROforma [76], and in general, they represent clinical guidelines as hierarchical plans that contain constructs for decisions and actions as well as embedded sub-plans [58]. The PROforma language, for example, distinguishes between Actions, Enquiries (which may be considered as a special type of action [58]), Decisions and Plans (which are collections of other tasks) [76]. Note that although rule-based languages have also been developed to formalise clinical knowledge, most notably the Arden Syntax [63] and the openEHR Guideline

Definition Language [84], they are considered out-of-scope as the formalised production rules cannot intrinsically capture the complexity of clinical guidelines, which typically involve multiple steps that unfold over time [58].

Regardless of the specific formalism, task-network models encapsulate the control flow (i.e. the logical ordering) between different tasks over time [58]. As a result, they assume a centralized system architecture in which a supervisory component controls the execution of guidelines. To reduce reliance on such components, Shalom et al. in 2015 proposed a projection mechanism whereby self-contained portions of a clinical guideline are identified for execution in parallel with the overall plan [70]. This mechanism was implemented in the MobiGuide system, which as described in Sec. 1.4 contains two decision support systems, a front-end system running on the patient's smartphone to execute guideline fragments locally; and a back-end system running on hospital servers to execute the overall guideline and "project" the appropriate portions to recipient devices [70]. While this projection mechanism can be extended to an arbitrary number of "local" devices, it still requires a supervisory component to project guideline fragments.

3.2.2 Data Flow Representations of Disease Management

Instead of modelling the control flow in clinical guidelines, a possible alternative is to focus on the data flow between each process. Such data flow models have been applied to represent disease management, a typical example of which is described by Carson, Cramp, Morgan and Roudsari in 1998 for the design and evaluation of clinical decision support systems [12]. Their model comprises a sequence of three main processes controlling the patient state: one to monitor the patient, another to make the appropriate clinical decisions and the third to effect the chosen decision. Their model was not formalised to enable the automatic execution of clinical guidelines; instead, it was applied conceptually to capture the different levels of disease management, from the business level to the patient level [12].

Similar data flow models have also been proposed for studying intelligent agents, which form one of the main approaches to research in artificial intelligence [62]. Defined as entities that can perceive their environment through sensors and act upon it via actuators, intelligent agents can be divided into four categories of increasing

complexity [62], from simple reflex agents that can only apply simple rules to determine the appropriate action given the current percept to utility-based agents that can not only maintain a conception about the state of the world but also maximise the expected utility of their actions.

Such agent-based approaches have also emerged as a new paradigm in software engineering [14], and in 2003, Kephart and Chess proposed a model comprising four processes, namely Monitor, Analyze, Plan and Execute, to represent the self-management of computing systems [38]. Lasierra, Alesanco and García (2014) then adapted the Kephart and Chess model for the healthcare domain, developing the HOTMES ontology for seamless integration of heterogeneous data in telemedicine systems [43].

3.3 Generic Model of Data Flow

3.3.1 Model of a Single Process

In contrast to the typical approaches for formalising clinical knowledge, this thesis focuses on capturing the data flow between different processes to remove the reliance on a centralised controller, whether for executing guidelines or projecting guideline fragments. In particular, guidelines are represented as a network of data flow processes executing in parallel. In this way, guideline knowledge and reasoning can be flexibly distributed across the components of a pervasive healthcare system such that they can provide decision support independently of each other, thereby achieving the "holonic" multi-agent architecture presented in Sec. 1.2.1.

More formally, a data flow process is modelled as a tuple containing:

- An ID (*Id*) to distinguish it from other processes.
- A data state (*DataState*) for storing previous data input into the process.
- A control state (*ControlState*) which determines when the process is activated.
- A specification (*InstSpec*) of the instructions for the process when it is activated.

$$Process = Id \times DataState \times ControlState \times InstSpec \tag{3.1}$$

Since this chapter focuses on the architecture model, the details of the data state, control state and instruction specification are left unspecified; it is sufficient to note that they fully determine how any input data is operated on by a process at any given time. In particular, processes are modelled to execute at every time instant (regardless of whether they are activated or not), and each execution comprises the following three steps:

- Generate new output data using the input data and current date-time.
- Update the data state of the process based on the input and output data as well as the current date-time.
- Update the control state of the process based on the input and output data as well as the current date-time.

Let the function *execute* represent the overall execution of a process, and let *generateData*, *updateDataState* and *updateControlState* represent the functions for each of the three steps. They have the following signatures and satisfy the following invariant:

$$execute : Process \times \mathscr{P}(Data) \times DateTime \rightarrow Process \times \mathscr{P}(Data) \tag{3.2}$$

$$\begin{aligned} generateData : {} & DataState \times ControlState \times InstSpec \times \mathscr{P}(Data) \\ & \times DateTime \rightarrow \mathscr{P}(Data) \end{aligned} \tag{3.3}$$

$$\begin{aligned} updateDataState : {} & DataState \times ControlState \times InstSpec \times \mathscr{P}(Data) \\ & \times DateTime \rightarrow DataState \end{aligned} \tag{3.4}$$

$$\begin{aligned} updateControlState : {} & Id \times DataState \times ControlState \times \mathscr{P}(Data) \\ & \times DateTime \rightarrow ControlState \end{aligned} \tag{3.5}$$

Invariant 3.1. Let p be an arbitrary process, d_{in} an input data set and t a date-time stamp. Furthermore, let $execute(p, d_{in}, t) = (p_{out}, d_{out})$. Then:

$$\pi_1(p_{out}) = \pi_1(p) \wedge \pi_4(p_{out}) = \pi_4(p) \wedge$$
$$\pi_2(p_{out}) = updateDataState(\pi_2(p), \pi_3(p), \pi_4(p), d_{in} \cup d_{out}, t) \wedge$$
$$\pi_3(p_{out}) = updateControlState(\pi_1(p), \pi_2(p), \pi_3(p), d_{in} \cup d_{out}, t) \wedge$$
$$d_{out} = generateData(\pi_2(p), \pi_3(p), \pi_4(p), d_{in}, t)$$

The specification of *updateDataState*, *generateData* and *updateControlState* depends on the specifics of the adopted formalism and will therefore not be detailed until subsequent chapters. However, it is useful to note that as implied by the invariant (Inv. 3.1), a process is assumed to never change its identifier nor its instructions, which intuitively means that it can never be transformed into other process. Indeed, the signatures of *updateDataState* and *generateData* reveal that the behaviour of a process is not affected by its ID; the process ID is required by *updateControlState* to ensure that only the relevant data can update its control state, while the rest of the process behaviour is specified completely by its instruction specifications.

By updating the control state independently of the instructions of a process, a clear separation of concerns is ensured between when a process is activated and what the process does when activated. In fact, let *isProcessActivated* be a function that returns a boolean indicating whether the process is activated or not. This function only depends on the control state of the process as well as the current datetime:

$$isProcessActivated : ControlState \times DateTime \rightarrow Boolean \tag{3.6}$$

If *isProcessActivated* returns false, then *generateData* should return an empty set of output data regardless of everything else. Conversely, if *isProcessActivated* returns true, the output of *generateData* should be independent of the process control state. This leads to the following two invariants:

Invariant 3.2. Let s_{data} be an arbitrary data state, s_{ctrl} a control state, s_{inst} an instruction specification, d_{in} an input data set and t a date-time stamp. Then:

$$\neg isProcessActivated(s_{ctrl}, t) \Rightarrow generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{\}$$

Invariant 3.3. Let s_{data} be an arbitrary data state, s_{ctrl1} and s_{ctrl2} two different control states, s_{inst} an instruction specification, d_{in} an input data set and t a date-time stamp. Then:

$$isProcessActivated(s_{ctrl1}, t) \wedge isProcessActivated(s_{ctrl2}, t) \Rightarrow$$
$$generateData(s_{data}, s_{ctrl1}, s_{inst}, d_{in}, t) = generateData(s_{data}, s_{ctrl2}, s_{inst}, d_{in}, t)$$

3.3.2 Model of a Process Network

While it is possible to model all disease management tasks as one conglomerate process, it is useful for system design and development to divide the tasks into a network of processes. For this thesis, which focuses only on the clinical aspects of disease management, it is assumed that the processes in the network are fully connected and that data flows instantaneously between them. Furthermore, the processes are modelled to execute in parallel; in other words, there is no notion of execution order between the processes. Thus a process network can simply be modelled as a set of processes:

$$ProcessNetwork = \mathscr{P}(Process) \tag{3.7}$$

Let the function *executeNetwork* represent the execution of a process network at a given time instant. It is analogous to *execute* for individual processes (Eq. 3.2) and has the following signature:

$$\begin{aligned} executeNetwork : ProcessNetwork \times \mathscr{P}(Data) \times DateTime \\ \rightarrow ProcessNetwork \times \mathscr{P}(Data) \end{aligned} \tag{3.8}$$

At each time instant, processes in a network output generate and share data with each other, which in turn can affect the data they generate. Therefore, during execution, the processes in a network are modelled to continue generating data until a closure is reached. Afterwards, based on all the generated data, all processes update their data state and control state before the next time instant. This leads to the following two invariants:

Invariant 3.4. Let p_{net} be an arbitrary process network, d_{in} an input data set and t a date-time stamp. Furthermore, let $(p_{out}, d_{out}) = executeNetwork(p_{net}, d_{in}, t)$, then:

$$d_{out} = \{generateData(\pi_2(p), \pi_3(p), \pi_4(p), d_{in} \cup d_{out}, t) \mid p \in p_{net}\}$$

Invariant 3.5. Let p_{net} be an arbitrary process network, d_{in} an input data set and t a date-time stamp. Furthermore, let $(p_{out}, d_{out}) = executeNetwork(p_{net}, d_{in}, t)$, then:

$$\begin{gathered} p_{out} = \{(\pi_1(p), \\ updateDataState(\pi_2(p), \pi_3(p), \pi_4(p), d_{in} \cup d_{out}, t), \\ updateControlState(\pi_1(p), \pi_2(p), \pi_3(p), d_{in} \cup d_{out}, t), \\ \pi_4(p)) \mid p \in p_{net}\} \end{gathered}$$

Note that *executeNetwork* does not invoke *execute* on individual processes. This ensures that at each time instant, each process can only exhibit one data state and one control state. However, it is trivial to show that for a single process, *executeNetwork* induces the same behaviour as *execute*, provided that the output data is not a relevant input to the process.

3.4 The Computation Independent Model

The definitions and invariants presented in Sec. 3.3 constitute a generic model of data flow processes which is to be specialised into a CIM and PIM for the clinical domain. In particular, the adopted CIM is based on those of Carson et al. (1998) [12] and Lasierra et al. (2014) [43] but adapts them for a different purpose, namely for facilitating the execution of clinical knowledge.

By definition, the CIM is independent of any technology platform. Therefore, unlike that of Lasierra et al. (2014), the CIM abstracts away the existence of computer systems and represents the complete disease management process at the conceptual level, with a clear distinction between the clinical semantics of each process and the manner in which each process is executed. Thus whereas the Monitor and Execute tasks are considered by Lasierra et al. (2014) to be the entry and exit points of their system respectively [43], the CIM does not assume specific points of entry or exit for

each type of task. For example, although the process of diagnosing a fever and the process of prescribing a medication may both require confirmation from the clinician, they are considered as two different types of processes in the CIM, namely that of diagnostic and therapeutic decision-making respectively.

More specifically, the CIM represents the disease management process as consisting of four processes controlling the environment as shown in Fig. 3.1: Monitoring (M), Analysis (A), Decision (D), and Effectuation (E). Rounded rectangles with solid borders represent processes within the CIM, while the rounded rectangle with dash borders represent an external entity (viz. the environment). Furthermore, the arrows represent data flow within the CIM, with inbound arrows indicating the types of data that are operated on by the process and output arrows indicating the types of data generated. Thus Fig. 3.1 shows, for example, that Monitoring processes operate on physical stimuli and control instructions to generate low-level concepts as output.

More formally, the CIM can be specified by defining the following subsets of the generic data flow model, with subscript $_{CIM}$ indicating that it is specific to the CIM:

$$\begin{aligned} ProcessNetwork_{CIM} &\subset ProcessNetwork \\ &= \mathscr{P}(Process_{CIM}) \qquad (3.9) \\ Process_{CIM} &\subset Process \\ &= Monitoring_{CIM} \cup Analysis_{CIM} \\ &\quad \cup Decision_{CIM} \cup Effectuation_{CIM} \qquad (3.10) \\ Data_{CIM} &\subset Data \\ &= PhysicalStimulus_{CIM} \cup LowLevelConcept_{CIM} \\ &\quad \cup HighLevelConcept_{CIM} \cup ActionPlan_{CIM} \\ &\quad \cup PhysicalAction_{CIM} \cup ControlInstruction_{CIM} \qquad (3.11) \end{aligned}$$

Furthermore, the CIM satisfies the following two invariants, which specify, respectively, that a process can only operate on and generate data of specific types (as shown in Fig. 3.1). Note that since these two invariants do not form part of the developed archetype and guideline language, they are not given a number:

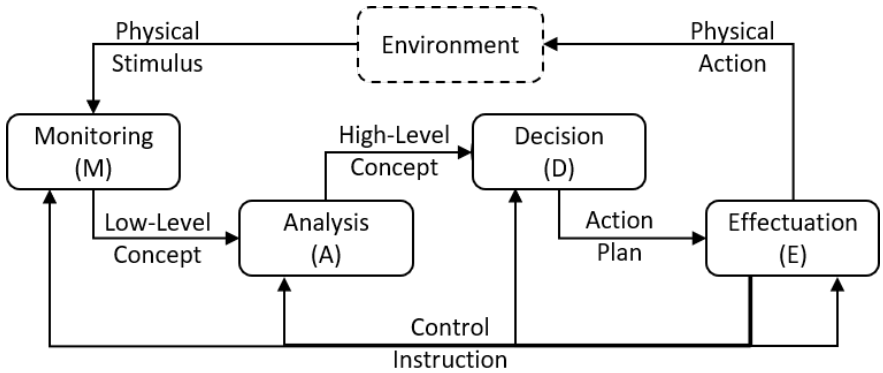

Fig. 3.1 The MADE computation independent model of disease management.

Invariant. Let P denote a specific process type (e.g. $Monitoring_{CIM}$), D_{in} the type of data it accepts (as shown in Fig. 3.1, such as $PhysicalStimulus_{CIM}$). Furthermore, let p be an arbitrary process in P, d_{in1} and d_{in2} two input data sets (not necessarily in D_{in}) and t a date-time stamp. Then:

$$d_{in1} \cap D_{in} = d_{in2} \cap D_{in} \Rightarrow$$
$$generateData(\pi_2(p), \pi_3(p), \pi_4(p), d_{in1}, t) =$$
$$generateData(\pi_2(p), \pi_3(p), \pi_4(p), d_{in2}, t)$$

Invariant. Let P denote a specific process type (e.g. $Monitoring_{CIM}$), D_{out} the type of data it outputs (as shown in Fig. 3.1, such as $LowLevelConcept_{CIM}$). Also, let p be an arbitrary process in P, d_{in} an input data set and t a date-time stamp. Then:

$$generateData(\pi_2(p), \pi_3(p), \pi_4(p), d_{in}, t) \subset D_{out}$$

Like the model of Carson et al. [12], the CIM includes processes for monitoring the patient and for performing the associated actions. However, the clinical decision making process has been partitioned into two separate processes, namely Analysis and Decision, to distinguish between diagnostic and therapeutic decision-making respectively. Furthermore, unlike either the Carson et al. or the Kephart and Chess model as adopted by Lasierra et al. [43], our MADE model accounts for the possi-

bility of the system to monitor both the state of the patient as well as the patient's surroundings and to perform actions which affect the patient state indirectly via changes in his or her surroundings or in the functioning of the system itself.

Thus the four processes (Monitoring, Analysis, Decision and Effectuation) are defined as follows:

- Monitoring (M) is the process of making observations about the environment, including both the internal as well as the external environment of the patient. This involves processing and assigning meaning to physical stimuli, transforming them to low-level concepts (i.e. data) about the state of the environment. For a diabetic patient for example, a Monitoring process may be to measure their blood glucose and physical exercise intensity levels regularly.
- Analysis (A) is the process of making abstractions from the low-level concepts, transforming them into high-level, more meaningful concepts (i.e. information) about the environment. In the clinical context, Analysis can therefore be seen as a diagnostic process, which may, in the simplest case, only involve a single assessment of the patient but can include on-going assessments as well [66]. Examples include determining from the measured data whether the patient is exhibiting good glycaemic control and is being compliant with the recommended physical exercise levels.
- Decision (D) is the process of deciding on the appropriate plan (i.e. course of action) given the current state of the environment and can therefore be seen as therapeutic decision-making. For example, if the patient has poor control over their blood glucose levels, then a Decision process may produce a recommendation to begin insulin therapy. In general, the Decision process also includes some scheduling as well as planning, as the resources required to execute the plan, such as sensors and actuators, may not be available. However, for simplicity, we will assume that these resources are always available.
- Effectuation (E) is the process of performing the decided course of action, with the intention of bringing about a change in the patient's state, such as administering insulin to reduce the patient's blood glucose levels. The course of action need not affect the patient's state directly; it may also involve changing the patient's external environment or controlling another MADE process, such

as changing the frequency at which the patient's state is analysed or the specific physical stimuli that are monitored.

Although the final aim of this thesis is to support the formalisation and execution of clinical knowledge, it should be noted that at the computation independent level, no assumption is made regarding the specific reasoning mechanism employed for each type of MADE process. For example, there are many different approaches to performing a diagnosis [94] which differ not only by their representation of knowledge, but also by their representation of time and state changes as well as by their inferencing method. However, despite these differences, they can all be conceptually modelled as an Analysis process as, by definition, the task of diagnosis is one of determining the appropriate explanations (high-level concepts) given the observations (low-level concepts) [94]. The only restriction is that each MADE process should conform to the definitions and invariants presented.

3.5 The Platform Independent Model

3.5.1 Overview

The CIM reveals that to fully support a patient, the knowledge-based system must be able to support all four types of processes (Monitoring, Analysis, Decision and Effectuation). However, to derive a knowledge representation from the CIM, a few changes are required to convert it to a corresponding PIM. For example, software systems cannot interact with the patient directly but can only instead interact through interfaces provided by external components. In other words, although user interfaces can comprise arbitrarily complex sensors, they cannot provide physical stimuli directly to a software system. Likewise, the system cannot execute physical actions directly but must rely on an interface with the appropriate actuators.

In fact, it may be the case that manual intervention is needed to complete a process or confirm the output of a process. For example, an action plan resulting from a decision may require verification by a clinician before it is executed by the system. Similarly, a complex analysis of patient data may be performed manually before being input into the system to make the appropriate decision. Thus taking these factors into account, the adopted platform independent model (PIM) for disease

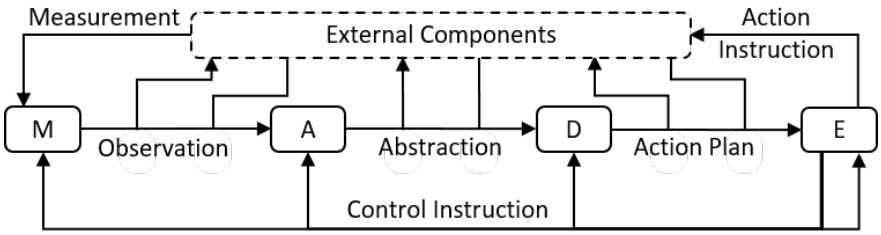

Fig. 3.2 The MADE platform independent model of disease management.

management is as shown in Fig. 3.2. This model can be formalised in an analogous manner to that for the CIM (Sec. 3.4); the main difference is that in the PIM, the Monitoring process takes measurements as inputs while the Effectuation process outputs action instructions. Furthermore, unlike the CIM, each process can send its outputs to and receive inputs from the user interface.

Although each type of process is depicted in Fig. 3.2 as a unified whole, they can in fact be decomposed into a set of parallel processes as shown in Fig. 3.3, with the dashed borders delineating the high-level MADE processes and the solid lines the decomposed sub-processes. This decomposition reflects the fact that the management of a disease often requires multiple, concomitant tasks to be performed in parallel. Thus although the MADE processes can be distributed en-bloc across the devices of the pervasive healthcare system, such that each process is completely performed by one (but not necessarily the same) device, it is also possible to distribute smaller sub-processes instead, thereby maximising the number of distribution options and the potential benefits of distribution.

For example, consider the guideline on obesity published by the National Institute for Health and Clinical Excellence (NICE) in the UK, which states that for adult patients, clinical intervention may involve some combination of diet, physical activity, drugs and surgery [52]. This decision may be modelled as four separate decisions for diet, physical activity, drugs and surgery respectively, each of which will be made based on a possibly overlapping set of patient information, such as the patient's physical ability and degree of obesity. The derivation of each piece of information can, in turn, be modelled as an individual Analysis process, and likewise, the performance of different measurements and the execution of different plans can be modelled

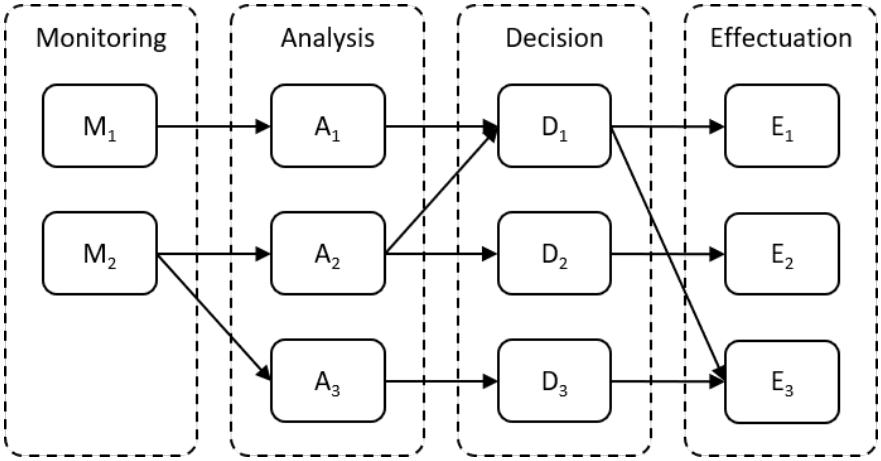

Fig. 3.3 The MADE platform independent model with parallel decomposition shown explicitly. For simplicity, the flow of control instructions is not shown.

as concurrent Monitoring and Effectuation processes respectively. As a result, the MADE processes will be decomposed in parallel as exemplified in Fig. 3.3.

3.5.2 Formal Specification

The overall MADE PIM can be formalised in an analogous manner to the MADE CIM presented in Sec. 3.4. Indeed, since all research outputs relate to the MADE PIM, no distinction is made in the remainder of this thesis between specifications of the generic data flow model and their specialisation to the MADE PIM. Therefore, unless otherwise indicated, the set *Process* for example shall henceforth refer to the set of all MADE processes in the PIM (instead of all generic data flow processes).

As implied in Fig. 3.2, four sub-types of MADE processes and six sub-types of MADE data are distinguished:

$$\begin{aligned} Process = {} & Monitoring \cup Analysis \\ & \cup Decision \cup Effectuation \end{aligned} \tag{3.12}$$

$$\begin{aligned} Data = {} & Measurement \cup Observation \\ & \cup Abstraction \cup ActionPlan \\ & \cup ActionInstruction \cup ControlInstruction \end{aligned} \tag{3.13}$$

Furthermore, analogous to the CIM, the PIM satisfies the following two invariants, which specify, respectively, that a process can only operate on and generate data of specific types:

Invariant 3.6. Let P denote a specific process type (e.g. *Monitoring*), D_{in} the type of data it accepts (as shown in Fig. 3.2, such as *Measurement*). Furthermore, let p be an arbitrary process in P, d_{in1} and d_{in2} two input data sets (not necessarily in D_{in}) and t a date-time stamp. Then:

$$\begin{gathered} d_{in1} \cap D_{in} = d_{in2} \cap D_{in} \Rightarrow \\ generateData(\pi_2(p), \pi_3(p), \pi_4(p), d_{in1}, t) = \\ generateData(\pi_2(p), \pi_3(p), \pi_4(p), d_{in2}, t) \end{gathered}$$

Invariant 3.7. Let P denote a specific process type (e.g. *Monitoring*), D_{out} the type of data it outputs (as shown in Fig. 3.2, such as *Measurement*). Also, let p be an arbitrary process in P, d_{in} an input data set and t a date-time stamp. Then:

$$generateData(\pi_2(p), \pi_3(p), \pi_4(p), d_{in}, t) \subset D_{out}$$

However, unlike the CIM, two categories of processes are distinguished: "proxy" processes and non-proxy processes. As mentioned in Sec. 3.5.1, certain processes may require manual intervention, thus they are differentiated from the automatable, non-proxy processes by their ID. More specifically, if a process has an ID in the set of proxy IDs (*ProxyId*), which is a subset of *Id*, then it is categorised as a proxy process that requires manual intervention. As such, all output data generated by that process cannot affect any other processes, which gives rise to the following invariant:

Invariant 3.8. Let p be an arbitrary processes, d_{in} an input data set and t a date-time stamp. Furthermore, let $(p_{out}, d_{out}) = execute(p, d_{in}, t)$, then:

$$\begin{gathered} \pi_1(p) \in ProxyId \Rightarrow \\ \forall p_2 \in Process,\ d_{in2} \in \mathscr{P}(Data),\ t_2 \in DateTime. \\ execute(p_2, d_{in2}, t_2) = execute(p_2, d_{in2} \cup d_{out}, t_2) \end{gathered}$$

"... we routinely monitor fasting urinary ketones [M] in women with GDM. The patient measures ketonuria using urine strips [M]. Monitor ketonuria routinely means to measure ketones in the urine every day at fasting conditions [M]. ... The results of ketonuria could be: a) positive (++); b) positive (+); c) negative (+/-); d) negative (-); e) negative (--). ... In case of ketonuria detection (the number of ketonuria measurements with result "positive" is equal or higher than 3 in a period of time of one week) [A]:- If the patient was COMPLIANT with the prescribed diet [A], the nurse decides to increase the carbohydrates intake either at dinner or at bedtime: the amount of carbohydrates at dinner or at bedtime is increased by 1 unit (10 grams [D]). ..."

Fig. 3.4 Extract from the MADE mark-up version of the GDM guideline [61] leading to a decision to increase carbohydrates intake. The MADE mark-up is indicated by underlining followed by inserted [M] for Monitoring, [A] for Analysis and [D] for Decision.

3.6 Application Example

It is outside the scope of this chapter to illustrate the use of the PIM on a full clinical guideline, but as a small example, consider the extract shown in Fig. 3.4 which is part of the narrative guideline adopted by MobiGuide for managing gestational diabetes mellitus (GDM); this fragment in particular relates to the decision to increase the carbohydrates intake of the GDM patient. In consultation with expert clinicians, a manual mark-up was performed on the extract to identify M, A, D and E processes as indicated in Fig. 3.4 by underlining followed by an inserted [M] for Monitoring, [A] for Analysis and [D] for Decision. In this fragment, no Effectuation processes were identified.

From the mark-up, a corresponding MADE process model as shown in Fig. 3.5 was constructed and was validated informally by the expert clinicians as an accurate representation of the knowledge in the original extract. The arrows in the figure depict the flow of data from one process to another, showing, for example, that the decision to increase carbohydrates intake depends on the analysis of fasting urinary ketone levels and carbohydrates intake levels. In this case, the specific logical condition for triggering the decision is the conjunction of detection of ketonuria and compliance to dietary prescription. Similarly, the figure also shows the required input and target output for the detection of ketonuria; underlying that at a lower-level

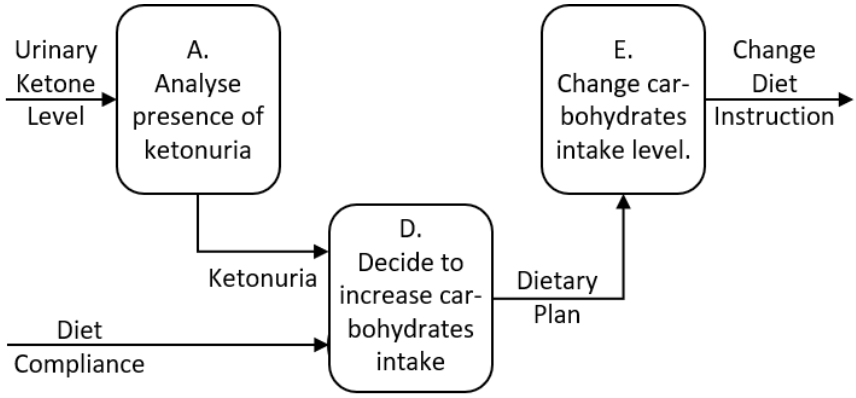

Fig. 3.5 Fragment of the MADE process model corresponding to the text extract from the GDM guideline shown in Fig. 3.4.

of abstraction (not shown) is the logical condition governing the Analysis process, which is presence of three or more positive ketonuria measurements in one week.

Note that Fig. 3.5 does not show the process to "monitor fasting urinary ketones" despite it being a Monitoring process conceptually. This is because the process is to be performed completely manually by the patient, thus it is considered external to the system (and therefore the model). As a result, Fig. 3.5 only shows urinary ketone levels being input into into the system, specifically into the process for analysing ketonuria. On the other hand, consultations with expert clinicians revealed that the decision to increase carbohydrates intake need not be performed manually by the nurse as indicated in Fig. 3.4. Therefore, the decision process can be automated and included in the Fig. 3.5.

Although shown graphically, this MADE model for the guideline extract can be formalised in the same manner as the MADE CIM and PIM, the results of which may then facilitate the identification of an appropriate distribution of processes across the pervasive healthcare system. For example, since the processes shown in Fig. 3.5 require patient interaction, they may be best deployed to a device that remains close to the patient (e.g. a smartphone). On the other hand, if a process is highly complex (which may be determined from its specification), or if a process requires interaction with a clinician, then it may be best deployed to a back-end server.

3.7 Discussion

3.7.1 Serial Decomposition of MADE Processes

While the complete results of formalizing the GDM guideline are reserved for subsequent chapters, it is useful at this point to review the high-level design decisions implied by the adopted PIM and CIM as they ultimately determine what can and cannot be represented. In particular, Fig. 3.3 implies that each type of MADE process (M, A, D and E) can only be decomposed into parallel processes but not serial processes. In other words, it has not been considered how these MADE processes might also be decomposed into a series of sub-processes by considering the sub-steps that constitute a MADE process.

For example, a diagnosis of influenza may be modelled as involving two steps, the first to analyse in parallel all the individual symptoms of influenza (e.g. fever, diarrhoea and sneezing [51]), and the second to analyse the combination of symptoms present. Likewise, a decision to start a drug therapy may be decomposed into a sequence of sub-steps for deciding on the exact drug dosage, administration frequency and the follow-up procedure. The result of such serial decomposition is shown schematically in Fig. 3.6, in which three parallel Analysis processes (delineated by the dashed borders) are modelled using two, three and one sub-processes respectively (delineated by the solid borders).

Although such serial decomposition enables more fine-grained representations of disease management, it is not incorporated into the PIM as it is hypothesized that such flexibility is unnecessary and introduces extra complexity to the model and ultimately to the developed knowledge representation language and related outputs. For example, different processes may be best decomposed into different numbers of serial processes, and the conceptual nature of each intermediate result may be different even for the same type of process (M, A, D or E), all of which must be modelled in the PIM.

3.7.2 Nested MADE Networks

As mentioned in Sec. 3.2, data flow models similar to the MADE CIM and PIM have been used to characterize the self-management of computer systems. This suggests

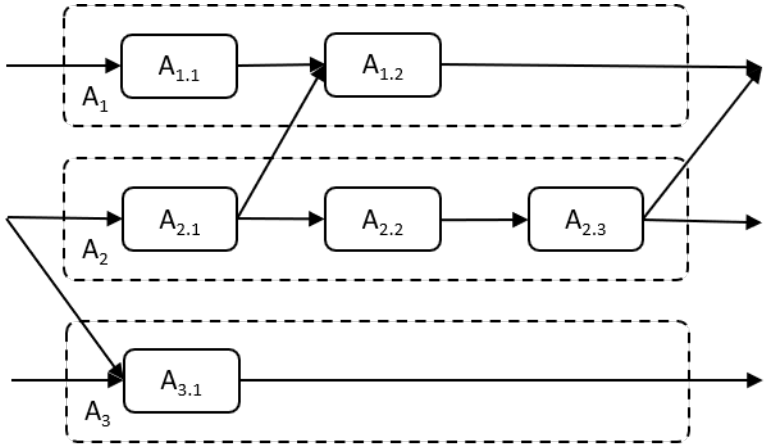

Fig. 3.6 Serial decomposition of the MADE model. For simplicity, only the Analysis processes are shown.

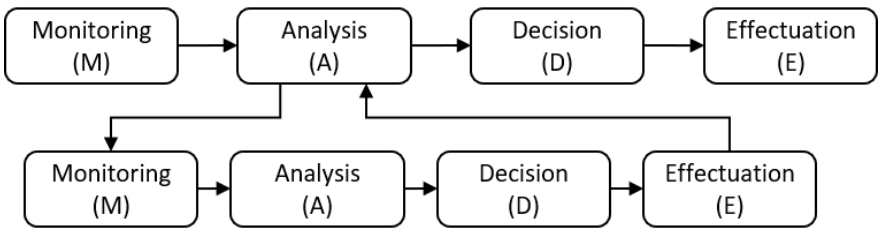

Fig. 3.7 Nesting of MADE networks. For simplicity, the external components are not shown.

the addition of nested MADE networks to the PIM, with the inner MADE networks controlling outer MADE networks. For example, as shown in Fig. 3.7, an Analysis process may itself be monitored, the results of which are then analysed and used to formulate a plan that is then acted upon to change the behaviour of the Analysis process.

Such nesting may be useful in modelling changes to the process due to changes in system performance. For example, in the event of connectivity issues between the different devices, the inner MADE network can adjust the functioning of the outer MADE network as appropriate. However, while useful, a clear separation of concerns should be maintained between the clinical aspects of maintaining a patient's health

and the technical aspects of maintaining the system behavior. The latter requires technical knowledge and is therefore be outside the scope of the PIM, which is designed to capture clinical processes.

Nevertheless, nested MADE networks need not be solely applied to addressing technical issues. Medically related issues, such as non-compliance to a treatment regime, should also be monitored and accounted for. An inner MADE network for diabetic patients may, for example, monitor their compliance to recommended diet and propose alternative treatments in case of deviations, such as administering insulin. However, it is hypothesised that such issues can also be fully modelled without use of nesting; compliance is simply a type of Analysis and the alternative treatments can form part of a Decision process. Thus like serial decomposition, support for nesting would simply introduce unnecessary complexity to the PIM and knowledge representation.

Chapter 4

The Reference Information Model

4.1 Introduction

The MADE PIM presented in Sec. 3.5 is designed to be both independent of any system-specific detail and applicable across different clinical applications, thus providing an appropriate starting point for developing the MADE knowledge representation language for pervasive healthcare systems. This requires formalizing the model to arrive at an appropriate syntax and semantics for the language, and this formalization is divided into two parts in this thesis: one part to model the data flowing between the different MADE processes and the other part to model the MADE processes themselves.

Indeed, to maximise their interoperability and adaptability, such separation of concerns between data and processes is generally accepted as good practice in systems that manage electronic health records (EHR). For example, since its establishment in 2003, the openEHR Foundation (https://www.openehr.org/) has proposed various standards for adaptable and interoperable EHR systems, including a reference model (RM) for representing clinical data [83] and a guideline definition language for enabling computerized clinical decision support [84]. Similarly, founded in 1987, Health Level Seven International (HL7, https://www.hl7.org/) has published standards such as the Arden Syntax for formalising clinical knowledge [63] and the virtual medical record (vMR) for representing data for clinical decision support [37].

This chapter focuses on the formal specification of the data flowing between MADE processes, which will be referred to as the MADE reference information model (RIM) and is analogous to the openEHR RM and HL7 vMR. Thus following a description of related work in Sec. 4.2, the complete specification of the MADE RIM is detailed in Sec. 4.3. To ensure unambiguity and to enable formal analysis, the specification is formalised using axiomatic set theory, the notation for which is summarised in Table 1.1. Furthermore, using the same example presented in Sec. 3.6, a simple application of the MADE RIM will be demonstrated in Sec. 4.4, while its validation using a complete clinical guideline is reserved for Ch. 8. Finally, the results are discussed at the end of the chapter (i.e. Sec. 4.5).

4.2 Related Work

To maximise re-use, EHR systems should generally be adaptable to different clinical applications, each of which may exhibit different requirements on clinical data. Therefore, these EHR systems generally adopt a multi-level modelling approach to representing clinical data, in which an application-independent model is implemented in the system and specialised during runtime. For example, openEHR distinguishes between its reference model, which is designed to capture the generic data types required in a clinical statement, and domain content models, which specialise the reference model for specific clinical applications by detailing their semantic constraints in the form of archetypes [83, 4]. Only the reference model is implemented in software, which allows the resulting system to be reused in different clinical applications, provided that the appropriate domain model is provided during run-time.

Similarly for HL7 v3, its reference information model is designed to represent all clinically relevant “messages”, which need not be limited to clinical decision support. Depending on the specific problem domain, the requirements for which are represented as a domain analysis model, the HL7 RIM may be specialised into different domain information models (DIM) [16]. Example domains include clinical genomics and emergency medical services, but of particular interest is the virtual medical record (vMR), which was developed to support clinical decision support and is comparable to the openEHR reference model.

As part of the MobiGuide project, the HL7 vMR model (first release) was adopted to support interoperability between the back-end servers of hospitals and the wearable healthcare technologies of patients. It was observed however [26, 45] that the HL7 vMR model was unable to capture all the necessary data requirements, and the solution of extending the vMR data types and relaxing their definitions was proposed and demonstrated. However, to best support the exchange of data in the special case of wearable and mobile healthcare technologies, this thesis proposes an alternative of developing a new RIM based specifically on the requirements of pervasive healthcare.

Similar to the openEHR RM and HL7 vMR, the new RIM is designed to be generic and to allow specialisation into the appropriate DIMs. However, the new RIM is also designed to be concise instead of comprehensive, since pervasive healthcare systems may be required to process large data streams (e.g. from sensors) on devices with limited computing capabilities (e.g. smartphones instead of back-end servers). For example, clinical observations in openEHR are modelled to contain data of arbitrary complexity (including lists, tables and trees), as well as optional state and protocol information (also of arbitrary complexity) [82]. Such detail and flexibility, while useful, may introduce unnecessary processing costs to the pervasive healthcare system as well as unnecessary complexity to the knowledge representation language.

Furthermore, neither the openEHR RM nor HL7 vMR explicitly models delayed and/or out-of-order arrival of data items, which may frequently occur in the pervasive healthcare setting and can affect the support provided to the patient. Indeed, by being based on the MADE PIM, the new RIM is also explicitly designed to reflect the data requirements of pervasive healthcare, such as the processing of low-level sensor data which are not explicitly modelled by the openEHR RM and HL7 vMR.

4.3 The MADE Data Types

As specified in Eq. 3.13, there are six main data types in the MADE RIM: measurements, observations, abstractions, action plans, action instructions and control instructions. They are each constructed from a combination of 10 primitive data types, namely *Id*, *Boolean*, *Nominal*, *Enumerated*, *Count*, *Proportion*, *Dimensioned*, *DateTime*, *Duration* and *Schedule*, all of which adopt conventional semantics. For

example, the data type Dimensioned represents quantities that are characterised by a real-number magnitude (e.g. 4.5) and a unit (e.g. mmol/L).

4.3.1 Measurements

One of the six high-level MADE data types represents measurements, which are data items that result from "an empirical estimation of an objective property or relation" [77] about the environment and may not have an immediate clinical interpretation. Examples include accelerometer, magnetometer and goniometer data from which measures of the patient's motor function can be derived [39]. In the MADE model, measurements are quantified using a dimensioned number and are associated with a valid date-time, which indicates when the measurement holds, i.e. the instant of measurement. Furthermore, as with other MADE data types, all measurements include an identifier to distinguish between different sensed physical stimuli as well as a transaction date-time which, as opposed to a valid date-time, indicates when the data item is known to the system. For example, if a patient's blood glucose level is measured at 10:00 but entered into the system two hours later at 12:00, then the measurement's valid and transaction time would be 10:00 and 12:00 respectively.

More formally, measurements are specified as follows:

$$\begin{aligned} Measurement = MeasurementType \times TransactionDateTime \\ \times ValidDateTime \times Dimensioned, \text{ where} \end{aligned} \tag{4.1}$$

$$MeasurementType = Id \tag{4.2}$$

$$TransactionDateTime = DateTime \tag{4.3}$$

$$ValidDateTime = DateTime \tag{4.4}$$

4.3.2 Observations

The result of processing measurements by a Monitoring process is an observation, which is a low-level fact about the environment that has a direct clinical interpretation but, as shown in Fig. 3.2, still requires further analysis for clinical decisions to be made. In the MADE model, two types of observations are distinguished: those for observed properties and those for observed events. This is analogous to the

distinction made by Shahar in 1997 between parameters and events in his framework for knowledge-based temporal abstractions [67], wherein parameters are defined as "a measurable aspect or a describable state of the world" and events as "the occurrence of an external volitional action or process".

$$Observation = ObservedProperty \cup ObservedEvent \tag{4.5}$$

Observed properties are observations about the physical objects in the environment, e.g. blood glucose level. Like measurements, observed properties are only valid at a specific time point, thus they also contain a single valid date-time stamp. However, they need not be quantified using dimensioned values. For example, blood type may be categorised into A, B, AB or O and is therefore a nominal data type, while burn severity may be assigned an ordered grade of either 1, 2, or 3 and is therefore an enumerated data type. Other possible data types for property values are boolean, count and proportion:

$$\begin{aligned} ObservedProperty &= PropertyType \times TransactionDateTime \\ &\times ValidDateTime \times PropertyValue, \text{ where} \end{aligned} \tag{4.6}$$

$$PropertyType = Id \tag{4.7}$$

$$\begin{aligned} PropertyValue &= Boolean \cup Nominal \cup Enumerated \\ &\cup Count \cup Proportion \cup Dimensioned \end{aligned} \tag{4.8}$$

The second type of observation (viz. observed event) relates to events which occur in the environment. These events may include [13]:

- Activities, which are homogeneous events with a duration but no natural end goal (e.g. running on a treadmill).
- Accomplishment, which is not homogeneous but has a culmination (e.g. climbing a mountain).
- Achievement, which is instantaneous and has a culmination (e.g. administering an insulin dosage).

- State, which is homogeneous and has a starting point but does not have a meaningful duration or culmination (e.g. a patient being diabetic).

Unlike observed properties, observed events in the MADE RIM can only be characterised by a Boolean value indicating whether the corresponding event occurred or not. All relevant properties of events, e.g. exercise intensity, are captured by the observed properties of the objects involved. Furthermore, all observations relating to events are associated with a pair of date-time stamps marking the start and end time of an event; in the simple case of instantaneous events, such as state transitions, these two date-time stamps are equal:

$$\begin{aligned} ObservedEvent &= EventType \times TransactionDateTime \\ &\times ValidDateTimeRange \times Boolean, \text{ where} \end{aligned} \tag{4.9}$$

$$EventType = Id \tag{4.10}$$

$$\begin{aligned} ValidDateTimeRange &= DateTime \times DateTime, \text{ such that} \\ &\forall v \in ValidDateTimeRange.\ \pi_1(v) \leq \pi_2(v) \end{aligned} \tag{4.11}$$

4.3.3 Abstractions

Abstractions result from the removal of irrelevant information from some given entities [11]. Thus compared with observations, abstractions in the MADE PIM may exhibit more abstract values and may be valid over an extended time period. For example, an abstraction of physical activity observations may compute a summary of patient's degree of activity over the month, while another abstraction of blood glucose observations may detect multiple hyperglycaemic episodes in the past week.

As a result, abstractions are modelled to also contain a valid date-time range and an abstraction value that can be of any numerical or categorical data type:

$$\begin{aligned} Abstraction &= AbstractionType \times TransactionDateTime \\ &\times ValidDateTimeRange \times AbstractionValue, \text{ where} \end{aligned} \tag{4.12}$$

$$AbstractionType = Id \tag{4.13}$$

$$AbstractionValue = Boolean \cup Nominal \cup Enumerated \cup Count \cup Proportion \cup Dimensioned \tag{4.14}$$

4.3.4 Action Instructions

To specify the action plans of a Decision process, it is useful to first specify the outputs of an Effectuation process (viz. action and control instructions) since they form the main constituents of an action plan. In general, an action is something that is done intentionally (as opposed to events which only occur) [89], and in the MADE model, two types of actions (and therefore action instructions) are distinguished: homogeneous actions and culminating actions. Note that for simplicity, actions and action instructions will not be distinguished where no ambiguity exists.

$$ActionInstruction = HomogeneousAction \cup CulminatingAction \tag{4.15}$$

Firstly, actions are referred to as homogeneous if they do not have a clear end goal and can be divided into sub-parts that retain the same overall properties, for example a continuous activity such as running on a treadmill. Therefore, the corresponding instructions are characterised by a starting date-time stamp, a rate at which the action should be performed (e.g. running at 7 kph) as well as a duration (e.g. for 20 minutes):

$$HomogeneousAction = ActionType \times TransactionDateTime \times StartDateTime \times Rate \times Duration, \text{ where} \tag{4.16}$$

$$ActionType = Id \tag{4.17}$$

$$StartDateTime = DateTime \tag{4.18}$$

$$Rate = Dimensioned \tag{4.19}$$

Unlike homogeneous actions, culminating actions have a clear end goal that must be achieved, such as administering 30 units of basal insulin in the evening. Therefore, although instructions for such actions also have a target start date-time, they exhibit a well-defined target goal state instead of a duration and rate:

$$\begin{aligned} CulminatingAction &= ActionType \times TransactionDateTime \\ &\times StartDateTime \times GoalState, where \end{aligned} \tag{4.20}$$

$$\begin{aligned} GoalState &= Boolean \cup Nominal \cup Enumerated \\ &\cup Count \cup Proportion \cup Dimensioned \end{aligned} \tag{4.21}$$

4.3.5 Control Instructions

Control instructions are used to determine when its target process should be activated. In the MADE model, all processes are assumed to be pre-existent and can only be re-scheduled and/or paused or resumed by a control instruction. Furthermore, to allow correct temporal ordering of multiple control instructions irrespective of any potential transaction delays, these control instructions are also characterised by a valid date-time stamp to indicate when they should be effected:

$$\begin{aligned} ControlInstruction &= TargetProcess \times TransactionDateTime \\ &\times ValidDateTime \times (Schedule \cup \{\text{NULL}\}) \times (Status \cup \{\text{NULL}\}), \text{ where} \end{aligned} \tag{4.22}$$

$$TargetProcess = Id \tag{4.23}$$

$$Schedule = RepeatPattern \times RepeatInterval \tag{4.24}$$

$$RepeatPattern = \mathscr{P}(DateTime) \tag{4.25}$$

$$RepeatInterval = Duration \cup \{\text{NEVER}, \text{ALWAYS}\} \tag{4.26}$$

$$Status = \{\text{PAUSED}, \text{RUNNING}\}, \text{ such that} \tag{4.27}$$

$$\forall c \in ControlInstruction.\ \neg(\pi_4(c) = \text{NULL} \wedge \pi_5(c) = \text{NULL})$$

As specified above, control instructions may contain a special NULL element in place of a new schedule or a new status for the target process, which indicates that the target process should continue to operate under its existing schedule or status as appropriate. However, since control instructions would serve no function if both their schedule and status are absent (i.e. if both are NULL), they must contain a new schedule or a new status for the target process or both.

Furthermore, in the MADE PIM, all processes run continually unless instructed otherwise by a control instruction. Thus a schedule in the MADE RIM comprises a possibly empty set of starting date-time stamps (i.e. the repeat pattern) which indicates when a process should start being activated as well as a duration (i.e. the repeat interval) which indicates how often the pattern should be repeated (if ever). For example, the starting pattern may dictate the process to start executing on Mar. 30, 2020 at 6:00 am, while the repeat interval may dictate the process to continually execute every week. Sentinel values NEVER and ALWAYS indicate that the target process should, respectively, never and continuously be activated at each time instant beyond its starting pattern.

4.3.6 Action Plans

In general, disease management may involve complex plans comprising sequential, parallel, iterative, unordered and/or cyclical activities [58, 65]. However, since pervasive telemedicine systems often operate continuously over long periods of time, they are modelled in this thesis to operate on cyclic parallel plans only, which, through the use of the appropriate schedules, can also be transformed into sequential and non-cyclical plans. Therefore, without loss of functionality, an action plan is modelled as containing a set of scheduled control and action instructions, each specifying when a MADE process or physical action should be executed:

$$\begin{aligned} ActionPlan = {} & PlanType \times TransactionDateTime \times ValidDateTime \\ & \times \mathscr{P}(ScheduledControl, ScheduledHomogeneousAction \\ & ScheduledCulminatingAction), \text{ where} \end{aligned} \tag{4.28}$$

$$PlanType = Id \tag{4.29}$$

$$\begin{aligned} ScheduledControl = {} & TargetProcess \times (Schedule \cup \{\text{NULL}\}) \\ & \times (Status \cup \{\text{NULL}\}) \end{aligned} \tag{4.30}$$

$$\begin{aligned} ScheduledHomogeneousAction = {} & ActionType \times Schedule \\ & \times Rate \times Duration \end{aligned} \tag{4.31}$$

$$ScheduledCulminatingAction = ActionType \times Schedule$$

$$\times \mathit{GoalState}, \text{ such that} \tag{4.32}$$
$$\forall p \in \mathit{ActionPlan}.\ (\forall i, j \in \pi_4(p).\ \pi_1(i) = \pi_1(j) \Rightarrow i = j) \wedge$$
$$\forall k \in \mathit{ScheduledControl}.\ \neg(\pi_2(k) = \textsc{Null} \wedge \pi_3(k) = \textsc{Null})$$

The first condition on an action plan ensures that each of its scheduled control and action instruction refers to a different target process or action type. In other words, there cannot be more than one instance of the same type of action or control instruction in the action plan, which prevents the occurrence of internal inconsistencies such as scheduling the same process simultaneously to always and never execute. Furthermore, like that for control instructions, the second condition ensures that scheduled control instructions must contain a new schedule or a new status or both.

4.4 Application Example

As an example application of deriving a domain information model from the MADE RIM, consider the clinical guideline fragment reproduced in Fig. 3.4 and represented in Fig. 3.5 as a MADE PIM. According to this fragment, patients with gestational diabetes mellitus (GDM) are to track their urinary ketone levels and compliance with their dietary prescription. If positive ketonuria (presence of urinary ketones) is detected and the patient has been compliant with her prescribed diet, then she may increase her carbohydrates intake once for subsequent dinners. Thus overall, the MADE RIM can be specialised into the following domain information model (DIM) for this guideline fragment, which comprises 5 data types, namely 1 observation, 2 abstractions, 1 action instruction and 1 action plan:

$$\mathit{UrinaryKetoneLevel} \subset \mathit{ObservedProperty}, \text{ such that} \tag{4.33}$$
$$\forall k \in \mathit{UrinaryKetoneLevel}.\ \pi_1(k) = \textsc{Urinary Ketone Level} \wedge$$
$$\pi_4(k) \in \{\text{- -}, \text{-}, \text{+/-}, \text{+}, \text{+ +}\}$$
$$\mathit{Ketonuria} \subset \mathit{Abstraction}, \text{ such that} \tag{4.34}$$
$$\forall k \in \mathit{Ketonuria}.\ \pi_1(k) = \textsc{Ketonuria} \wedge \pi_4(k) \in \{\textsc{Positive}\}$$

$$\mathit{DietCompliance} \subset \mathit{Abstraction}, \text{ such that} \tag{4.35}$$
$$\forall c \in \mathit{DietCompliance}.\ \pi_1(c) = \textsc{Diet Compliance}$$
$$\wedge\ \pi_4(c) \in \{\textsc{Compliant}\}$$
$$\mathit{DietaryPlan} \subset \mathit{ActionPlan}, \text{ such that} \tag{4.36}$$
$$\forall p \in \mathit{DietaryPlan}.\ \pi_1(p) = \textsc{Dietary Plan}$$
$$\wedge\ \exists i \in \pi_4(p).\ i \in \mathit{ScheduledCulminatingAction}$$
$$\wedge\ \pi_1(i) = \textsc{Change Diet Instruction}$$
$$\mathit{ChangeDietInstruction} \subset \mathit{CulminatingAction}, \text{ such that} \tag{4.37}$$
$$\forall i \in \mathit{ChangeDietInstruction}.\ \pi_1(i) = \textsc{Change Diet Instruction}$$

The values for urinary ketones, ketonuria and carbohydrates sufficiency are inferred from the original guideline text shown in Fig. 3.4. For example, it is stated that the results of measuring urinary ketones can be: “a) positive (++); b) positive (+); c) negative (+/-); d) negative (-); e) negative (- -)”. Furthermore, while it is unclear from the extract what is the complete set of action and control instructions that “increase carbohydrates intake” entail, the plan at least involves the action to increase carbohydrates intake, which is a culminating action. Hence the existential quantifier $\exists$ is used to specify that each instance of this action plan must at least contain such a scheduled culminating instruction. To effectuate this change, there is a need for a corresponding specification for the culminating action instruction.

4.5 Discussion

4.5.1 Appropriateness of the MADE RIM

A key premise for developing this MADE RIM instead of adopting an existing one is that the MADE RIM may be more appropriate for use in pervasive healthcare systems. In particular, unlike typical reference models such as those for openEHR and HL7 presented in Sec. 4.2, the MADE RIM is based on the MADE PIM for pervasive telemedicine systems and is explicitly designed to be concise and to distinguish between transaction and valid datetimes. However, while it is not the aim of this

chapter to provide clinical validation of the MADE RIM, it can be noted that one missing feature from the MADE RIM is quality-of-data (QoD) awareness properties.

Due to the uncontrolled nature of the daily living environment, clinical data may suffer from noise and many other sources of error, including measurement errors due to movement artefacts or from incorrectly positioned sensors. Therefore, to ensure the best clinical support for the patient, such potential inaccuracies should be taken into account when processing data items, such as by using quantified probabilities and utility functions or by means of a multi-dimensional measurement of QoD such as that proposed by [42] which encompasses accuracy, dependability, cost, timeliness and quality-of-evidence. That said, quality awareness reflects a technical concern and should therefore be separated from the clinical concerns of pervasive healthcare systems, which is the focus of this thesis.

4.5.2 Interoperability with Clinical Information Systems

Since the MADE RIM is designed specifically for pervasive telemedicine systems, it does not aim to replace the function of existing standards in clinical information systems, including openEHR and HL7. As a result, in order to store the MADE data items in such systems, which may be necessary due to legal and ethical reasons for example, a systematic procedure must also be developed for translating between the MADE RIM and standards in use.

A comparison between the specifications of the different models reveals that the MADE data types can be mapped onto the existing openEHR RIM and HL7 vMR DIM as shown in Table 4.1. Although there is not a one-to-one mapping between the data types, the openEHR and HL7 data types can still distinguish between all MADE data types through judicious use of their attributes. However, it has been shown [26, 45] that certain extensions are nevertheless required to capture all the data requirements in pervasive healthcare systems. For example, [26, 45] demonstrated that transaction times should be added to vMR clinical statements; this holds also for openEHR classes.

Table 4.1 The correspondence between the data types in the MADE RIM and those in the openEHR RIM and HL7 vMR.

MADE RIM	**openEHR RIM**	**HL7 vMR**
Measurement	Observation	Observation Result
Observation		
Abstraction	Evaluation	
Action Plan	Instruction	Observation Order,
Action Instruction	Activity	Procedure Order and/or
Control Instruction		Substance Administration Order

Chapter 5

The Archetype Language

5.1 Introduction

While the MADE RIM presented in the previous chapter may be sufficient for representing all types of clinical data, it does not support the specification of well-formedness constraints on the data items. For example, it is possible to assign a value of $15\,\mathrm{ms}^{-2}$ to a body temperature, or a value of -73 °C to a prescribed target level of carbohydrates intake, both of which clearly lack any valid clinical interpretation. To capture these restrictions, OpenEHR extends its Reference Model with the notion of an archetype, which is "a computable definition, or specification, for a single, discrete clinical concept" [44]. Similarly, the HL7 standard relies on the use of templates, each of which is "an expression of a set of constraints on the RIM which is used to apply additional constraints to a portion of an instance of data" [32].

For the MADE RIM, an archetype language is developed and presented in this chapter to specify these well-formedness constraints on the MADE data items, which may then be used to check the validity of any instantiated MADE data item. The syntax for the language is specified using the extended Backus-Naur form (EBNF) [34], although for simplicity, white space separating the EBNF elements are assumed and will not be explicitly specified. Furthermore, as summarised in Table 5.1, certain syntactic constructs from Rosette will be re-used, all of which are written in all capitals to distinguish from other syntactic elements. Indeed, since the language is implemented as a set of libraries on top of Rosette, it supports all

Table 5.1 Summary of the Rosette syntax that may appear in the EBNF expressions.

EBNF Element	Corresponding Rosette Construct
`ID`	An identifier, e.g. `x`.
`SYMBOL`	A symbol, i.e. an arbitrary atomic value, the syntax for which is the same as that for `ID` but preceded with `'`. E.g. `'x`.
`LAMBDA-EXPR`	A lambda expression, which creates a function. E.g. `(lambda (x) (+ x 1))`.
`NATURAL`	A natural number. E.g. `1`, `2`, `3`, etc.
`BOOLEAN`	A boolean value (`#t` for true and `#f` for false)
`RATIONAL`	A rational number. E.g. 1.2, 2.5, etc.
`EXPR`	An arbitrary expression.

constructs that are inherently provided by Rosette as well as those that arise from the implementation of the language.

In Sec. 5.2, the complete syntax of the MADE archetype language is presented along with an informal description of its semantics and examples of its usage. This is followed by an explanation of its reference implementation in Sec. 5.3. A mixture of testing and formal verification is used to verify the implementation, the process and results for which is described in Sec. 5.4. Finally, this chapter concludes with a discussion in Sec. 5.5.

5.2 Language Specification

5.2.1 Basic Data Types

Since the MADE RIM is built on top of 10 primitive data types, namely *Id*, *Boolean*, *Nominal*, *Enumerated*, *Count*, *Proportion*, *Dimensioned*, *DateTime*, *Duration* and *Schedule*, expressions in the archetype language may involve manipulating and instantiating them. This sub-section focuses on the 7 non-temporal, basic data types, i.e. *Id*, *Boolean*, *Nominal*, *Enumerated*, *Count*, *Proportion*, *Dimensioned*, while the

next sub-section focuses on the remaining three temporal data types, i.e. *DateTime*, *Duration* and *Schedule*.

Identifiers are inherently supported by Rosette, thus for simplicity, no additional syntactic forms for constructing and manipulating Ids are provided by the MADE archetype language. For the 6 other non-temporal primitive data types, they can be instantiated in the same manner as any other structure in Rosette. More specifically, instantiations of the these primitive data types exhibit the following forms:

```
boolean-instance : '(bool', EXPR, ')';
nominal-instance : '(', ID, EXPR, ')';
enumerated-instance : '(', ID, EXPR, ')';
count-instance : '(count', EXPR, ')';
proportion-instance : '(proportion', EXPR, ')';
dimensioned-instance : '(dimensioned', EXPR, EXPR, ')';
```

As expected, `EXPR` should return a value of the appropriate type when instantiating a primitive data type. For example, for boolean instances, `EXPR` should always evaluate to true or false when executed. Furthermore, since Nominal and Enumerated each represent a family of data types, the members of which can only be determined when formalising a clinical guideline, no specific identifiers are associated with them. This ensures, for example, that a 'high' blood glucose level can be interpreted differently than a 'high' urinary ketone level, even though their values are both 'high'. Finally, for dimensioned instances, a second expression is required to produce a symbol that represents its units of measurement.

As with all other structures in Rosette, instances of these basic data types can be compared for equality using the `eq?` procedure. Additionally, the procedures `get-type`, `get-value` and `valid?` are provided by the MADE archetype language, which respectively returns the type of the instance, the value of its expression, and whether its value is of the appropriate type. Example invocations of the procedures are as follows. Note that for simplicity, unit conversion is not supported, thus in the MADE archetype language, 1000 m is not equal to 1 km.

```
> (eq? (dimensioned 1000 'm) (dimensioned 1 'km))
#f
```

```
> (eq? (count (- 11 2)) (count (* 3 3)))
#t
> (get-type (bool #t))
#<procedure:bool>
> (get-value (proportion (+ 3 1)))
4
> (valid? (bool 4))
#f
> (valid? (proportion (+ 3 1)))
#t
```

For enumerated and dimensioned instances, comparators are also supported, specifically `enum>?` and `enum<?` for enumerated instances and `dim>?`, `dim>=?`, `dim<?` and `dim<=?` for dimensioned instances. Example invocations of these procedures are as follows, where `test-enum` is an enumerated data type with values `'low`, `'normal` and `'high`:

```
> (enum>? (test-enum 'low) (test-enum 'high))
#f
> (enum<? (test-enum 'low) (test-enum 'high))
#t
> (dim>? (dimensioned 5 'm) (dimensioned 5 'm))
#f
> (dim>=? (dimensioned 5 'm) (dimensioned 5 'm))
#t
> (dim<? (dimensioned 1 'm) (dimensioned 15 'm))
#t
> (dim<=? (dimensioned 1 'm) (dimensioned 15 'm))
#t
```

5.2.2 Temporal Data Types

The temporal data types (i.e. *DateTime*, *Duration* and *Schedule*) can be instantiated and manipulated in a similar manner as the basic data types presented above, except

that they exhibit a more complex form and supports more complex operations. Firstly, duration instances comprise four fields representing the number of days, hours, minutes and seconds in the duration:

```
duration-instance : '(duration', EXPR, EXPR, EXPR, EXPR, ')';
```

The procedures `dur+` and `dur-` are provided for adding and subtracting two durations. Furthermore, to compare whether one duration is equal, larger than, and less than another, the procedures `dur=?`, `dur>?` and `dur<?` are provided. Note that these comparators can take into account the equivalence between different units of time, such that 1 day is equal to 24 hours and 1 hour is equal to 60 minutes, etc. In particular, this means that `dur=?` is not equivalent to `eq?`, as the latter compares each individual field for equality. Some examples of invoking these procedures are as follows:

```
> (dur+ (duration 1 0 0 0) (duration 0 30 0 15))
(duration 1 30 0 15)
> (dur- (duration 1 30 0 15) (duration 0 30 0 15))
(duration 1 0 0 0)
> (dur>? (duration 1 0 0 0) (duration 0 30 0 15))
#f
> (dur<? (duration 1 0 0 0) (duration 0 30 0 15))
#t
> (dur=? (duration 1 0 0 0) (duration 0 23 59 60))
#t
> (eq? (duration 1 0 0 0) (duration 0 23 59 60))
#f
```

Date-time instances comprise six fields representing the year, month, day, hour, minute and second.

```
datetime-instance :
  '(datetime', EXPR, EXPR, EXPR, EXPR, EXPR, EXPR, ')';
```

Durations can be added and subtracted from date-time instances using the procedures `dt-` and `dt+`. Furthermore, two date-time instances can be compared for

equality, greater than and less than using the procedures `dt=?`, `dt>?` and `dt<?`. For convenience, the comparators `dt>=?` and `dt<=?` are also supported. Note that these operators on date-time stamps can adjust for carry-overs, such that subtracting 1 day from March 1 would, for example, return Februrary 29 on leap years and February 28 otherwise. Some examples of invoking these procedures are as follows:

```
> (dt- (datetime 2020 3 1 15 0 0) (duration 1 0 0 0))
(datetime 2020 2 29 15 0 0)
> (dt+ (datetime 2020 2 29 15 0 0) (duration 1 0 0 0))
(datetime 2020 3 1 15 0 0)
> (dt>? (datetime 2020 3 1 0 0 0) (datetime 2020 2 29 24 30 0))
#f
> (dt<? (datetime 2020 3 1 0 0 0) (datetime 2020 2 29 24 30 0))
#t
> (dt=? (datetime 2020 3 1 0 30 0) (datetime 2020 2 29 24 30 0))
#t
```

Recall that schedules are defined in the MADE RIM as follows (Eq. 4.24 to Eq. 4.26):

$$\mathit{Schedule} = \mathit{RepeatPattern} \times \mathit{RepeatInterval}$$
$$\mathit{RepeatPattern} = \mathscr{P}(\mathit{DateTime})$$
$$\mathit{RepeatInterval} = \mathit{Duration} \cup \{\textsc{Never}, \textsc{Always}\}$$

This translates to the following syntactic form for schedules in the MADE archetype language, which contains two expressions:

```
schedule-instance : '(schedule', EXPR, EXPR, ')';
```

The first expression must evaluate to a list of date-time instances, while the second must evaluate to a duration or a boolean value, where true represents ALWAYS and false NEVER. For example, executing the following line of code creates a schedule (with identifier `test-schedule`) that starts on March 15, 2020 at 17:00:00 and March 19, 2020 at 15:30:00, repeating weekly:

```
> (define test-schedule
    (schedule (list (datetime 2020 3 15 17 0 0)
                    (datetime 2020 3 19 15 30 0))
              (duration 7 0 0 0)))
```

Unlike durations and date-time stamps, operations for manipulating and comparing schedules are not supported by the MADE archetype language. Instead, the procedure `on-schedule?` is provided that checks whether an input date-time stamp lies on the input schedule or not. For example:

```
> (on-schedule? test-schedule (datetime 2020 3 12 15 30 0))
#f
> (on-schedule? test-schedule (datetime 2020 3 15 17 0 0))
#t
> (on-schedule? test-schedule (datetime 2020 3 22 17 0 0))
#t
> (on-schedule? test-schedule (datetime 2020 3 22 17 30 0))
#f
```

5.2.3 Measurement Archetypes

Types of measurements are specified using the following syntactic form:

```
measurement-archetype :
  '(define-measurement', ID, units, [LAMBDA-EXPR], ')';
units : SYMBOL;
```

`ID` is the identifier for the specified measurement type, while `units` specify the measurement's unit of measure. Furthermore, the optional lambda expression specifies an arbitrary invariant on the measurement value; it must return true or false. For example, the following syntax creates an archetype for body temperatures that is measured in °C and accepts any numerical value larger than or equal to -273 (absolute zero). Note that for simplicity, each measurement archetype can only have one specific unit of measurement; convsersions, such as from °C to °F, are not supported.

```
> (define-measurement body-temperature 'celsius
    (lambda (d) (dim>=? d (dimensioned -273 'celsius))))
```

5.2.4 Observation Archetypes

As specified in the MADE RIM, observations can relate to either events or properties. For observed events, which can only take Boolean values by definition, the sole purpose of their archetypes is to assign the appropriate identifiers. As a result, all archetype specifications for observed events exhibit the following syntactic form, in which the string `#:event` is a keyword to indicate that the observation archetype is for observed events:

```
observed-event-archetype :
  '(define-observation', ID, '#:event)';
```

To specify archetypes of observed properties, the syntactic form is as follows:

```
observed-property-archetype :
  '(define-observation', ID, type, [LAMBDA-EXPR], ')'
  | '(define-observation', ID, 'nominal', value, {value}, ')'
  | '(define-observation', ID, 'enumerated', value, {value}, ')'
  | '(define-observation', ID, 'dimensioned', units,
      [LAMBDA-EXPR], ')';
type : 'bool' | 'count' | 'proportion';
value : SYMBOL;
```

As with measurement archetypes, `ID` is the identifier for the specified observed property type, while the optional lambda expression is an arbitrary invariant over the property's value. `type` specifies the type of values (e.g. dimensioned, Boolean, etc.) that can be associated to instances of ID. If a nominal value is expected, then a non-empty list of acceptable values must be provided; the same is true for enumerated values, but in this case the list must also be ordered (from least to most). If a dimensioned value is expected, then the units of measurement must be provided.

For example, the following syntax creates an archetype for an observation about the severity of a burn (ordered from 1st which is least severe to 4th which is most severe):

```
> (define-observation
    burn-severity enumerated '1st '2nd '3rd '4th)
```

Note that whenever an archetype is specified to contain nominal or enumerated values, an appropriate ID is automatically provided by the MADE archetype language to instantiate those nominal or enumerated data items. More specifically, if `id` is the ID for the specified archetype, then `id-value-space` would be the identifier for the nominal or enumerated data items. Thus the value for a 3rd degree burn would be instantiated as follows:

```
> (burn-severity-value-space '3rd)
```

5.2.5 Abstraction Archetypes

Specifications for abstraction archetypes have the same syntactic form as those for observed properties, except that the procedure is `define-abstraction` instead of `define-observation`:

```
abstraction-archetype :
  '(define-abstraction', ID, type, [LAMBDA-EXPR], ')'
  | '(define-abstraction', ID, 'nominal', value, {value}, ')'
  | '(define-abstraction', ID, 'enumerated', value, {value}, ')'
  | '(define-abstraction', ID, 'dimensioned', units,
        [LAMBDA-EXPR], ')';
```

For example, an abstraction archetype to indicate whether a patient has exercised for a sufficient amount or not may be specified as follows:

```
> (define-abstraction exercise-sufficiency bool)
```

5.2.6 Action Plan Archetypes

Specifications for action plan archetypes comprise an identifier followed by one or more lists of targets indicating the scheduled instructions that may be contained within the action plan. For lists of the form `(homogeneous-action ...)` and `(culminating-action ...)`, the targets denote the permitted action types of

scheduled homogeneous actions and scheduled culminating actions respectively. For lists of the form `(control ...)`, the targets denote the permitted target processes of scheduled control instructions. Formally, using EBNF notation:

```
action-plan-archetype :
  '(define-action-plan', ID, target-list, {target-list}, ')';
target-list:
  homogeneous-action-list | culminating-action-list
  | control-list;
homogeneous-action-list :
  '(homogeneous-action', target, {target}, ')';
culminating-action-list :
  '(culminating-action', target, {target}, ')';
control-list : '(control', target, {target}, ')';
target : SYMBOL;
```

For example, an archetype for penicillin prescriptions may be specified as follows:

```
> (define-action-plan penicillin-prescription
    (culminating-action 'administer-penicillin))
```

5.2.7 Action Instruction Archetypes

Two types of action are defined in the MADE RIM: homogeneous and culminating. For homogeneous actions, which contain a dimensioned rate and a duration, the archetype specification has a similar form as that for measurements, but with the addition of the keyword `#:homogeneous` to distinguish it from archetypes for culminating actions. Furthermore, the lambda expression, if present, accepts two arguments instead of one (namely the rate and the duration of the action):

```
homogeneous-action-archetype :
  '(define-action-instruction', ID, '#:homogeneous', units,
    [LAMBDA-EXPR], ')';
```

For culminating actions, which contain a goal state, the archetype specification has a similar form as that for observed properties and abstractions, but with the additional keyword `#:culminating`:

```
culminating-action-archetype :
  '(define-action-instruction', ID, #:culminating, type,
    [LAMBDA-EXPR], ')'
  | '(define-action-instruction', ID, #:culminating,
        'nominal', value, {value}, ')'
  | '(define-action-instruction', ID, #:culminating,
        'enumerated', value, {value}, ')'
  | '(define-action-instruction', ID, #:culminating,
        'dimensioned', units, [LAMBDA-EXPR], ')';
```

An archetype for administering penicillin may, for example, be specified as follows:

```
> (define-action-instruction administer-penicillin #:culminating
      dimensioned 'mg)
```

5.2.8 Control Instruction Archetypes

Specifications for control instruction archetypes comprise an ID and a non-empty list of symbols indicating the permissible target processes:

```
control-instruction-archetype :
  '(define-control-instruction', ID, target, {target}, ')';
```

For example, an archetype for controlling decisions relating to anti-inflammatory prescriptions (e.g. ibuprofen and aspirin) may be specified as follows:

```
> (define-control-instruction control-anti-inflammatory-meds
      'prescribe-ibuprofen 'prescribe-aspirin)
```

5.2.9 Archetype Verifier

Apart from the syntactic forms presented above, the MADE archetype language also provides two extra solver-aided syntactic forms to support the verification of clinical guidelines: a verifier presented in this section and a getter in the next. The verifier, `verify-archetype`, is a procedure that checks whether the input archetype (e.g. `id`) is logically consistent or not:

```
> (verify-archetype id)
```

If the specification of `id` does not contain logical inconsistencies, then it returns an example instance of the archetype that satisfies its specification; otherwise it returns `(unsat)`, meaning no satisfiable instance can be found. For example, the following code checks the logical consistency of the `burn-severity` archetype, the result for which is an instance of `burn-severity`, specifically a 1st degree burn observed on Dec. 15, 2019 at 00:00:00:

```
> (verify-archetype burn-severity)
Example #<procedure:burn-severity>:
#(struct:burn-severity #f #(struct:datetime 2019 12 15 0 0 0)
#(struct:burn-severity-value-space 1st))
```

The result indicates that the specification, although not necessarily valid in the clinical context, is at least satisfiable. In contrast, the following specification of room temperature, which requires its value to be both greater than 100 and lower than 0°C, is unsatisfiable, such that executing `verify-archetype` would return `(unsat)`:

```
> (define-observation room-temperature dimensioned 'celsius
      (lambda (d) (and (dim>? d (dimensioned 100 'celsius))
                       (dim<? d (dimensioned 0 'celsius)))))
> (verify-archetype room-temperature)
Example #<procedure:room-temperature>:
(unsat)
```

5.2.10 Archetype Getter

For every archetype specified, a getter is also provided that will generate a symbolic instance of that archetype, i.e. an instance of that archetype with placeholders that can subsequently be populated by Rosette with concrete values. Each getter, `get-id`, where `id` is a placeholder for the archetype's identifier, exhibits at least one of the following syntactic forms:

```
symbolic-archetype-instance :
  '(get-', ID, [2 * datetime], ')'
```

```
  | '(get-', ID, 2 * [2 * datetime], ')'
  | '(get-', ID, [2 * datetime], {target}, ')'
  | '(get-', ID, [2 * datetime], [schedule-length], ')';
schedule-length : NATURAL;
```

For all archetype specifications, the getter accepts two optional date-time stamps as input to indicate the range of permissible values for the year, month, day, hour, minute and second of all date-time stamps of the generated archetype instance. If the date-time pair is not provided, then by default all date-time stamps are fixed to Dec. 15, 2019 `hr`:00:00, with the value of `hr` ranging from 0 to 23 inclusive. For observed event and abstraction archetypes, which are valid over a date-time range, a second pair of date-time stamps can be provided such that the valid start date-times and valid end date-times can range over different values.

Apart from the date-time pair, getters for action plan archetypes also accept as input a list of targets, which specifies the scheduled instructions that are contained within the generated symbolic instance of the action plan. If no list is provided, then the generated action plan will contain a scheduled instruction for all targets specified in the archetype. Indeed, getters for action plan archetypes also assume the specification of the corresponding action instructions. Finally, for control instruction archetypes, an optional natural number can be provided to the getter to specify the number of starting date-time stamps contained within its schedule.

As an example, the following is a symbolic instance of `burn-severity` created by executing `get-burn-severity`:

```
> (get-burn-severity)
(get-burn-severity)
(burn-severity
 #f
 (datetime 2019 12 15 dt-part$0 0 0)
 (burn-severity-value-space
  {[(= 0 enum-val$0) 1st] [(= 1 enum-val$0) 2nd]
  [(= 2 enum-val$0) 3rd] [(= 3 enum-val$0) 4th]}))
```

As expected, the datetime of the observation is set to the default value, with `dt-part$0` being a symbolic integer for the valid hour of the observation. Sim-

ilarly, `enum-val$0` is a symbolic integer that, depending on its value, determine whether the observation is a 1st degree burn (`enum-val$0` equals 0), 2nd degree burn (`enum-val$0` equals 1), etc. Note that the boolean value on line 2 (in this with value `#f`) is an extra flag on all instances of MADE data to indicate whether they are proxy or not, i.e. whether they can be used directly by the MADE processes or not. Although this flag is not part of the MADE RIM and is unnecessary, since proxy and non-proxy data can be given different Ids, it is introduced to simplify the process of specifying the MADE archetypes, as it avoids the need to specify two different archetypes to represent the same clinical data, one for non-proxy instances and the other proxy.

5.3 Reference Implementation

5.3.1 Implementation of the Data Types

Implementing the archetype language in Rosette involves implementing the constructs necessary for the MADE RIM as well as the appropriate macros to transform the language syntax into the appropriate Rosette constructs. The reader is referred to the source code (available at https://github.com/nlsfung/MADE-Language/tree/master/rim) for the full details of the implementation, but to summarise, each data type specified in the MADE RIM is implemented as a structure in Rosette (Fig. 5.1), which in many ways is analogous to a class in object-oriented programming. More specifically, a structure in Rosette is a record that can contain multiple fields, and it supports inheritance as well as interfaces.

For example, as shown at the bottom of Fig. 5.1, all data structures for the MADE RIM implement the typed interface. Since Rosette does not inherently support type-checking, the typed interface provides two methods for type-checking purposes: `get-type`, which returns the structure type of the input instance, and `valid?`, which checks whether the input instance satisfies some criteria specific to its type. For example, executing `valid?` on a date-time stamp checks whether, amongst other things, all fields are integers, the month ranges from 1 to 12 and the day from 1 to 28, 29, 30 or 31 depending on the year and month. Similarly, the `valid?` method for action plans checks, for example, whether it contains a list of scheduled instructions.

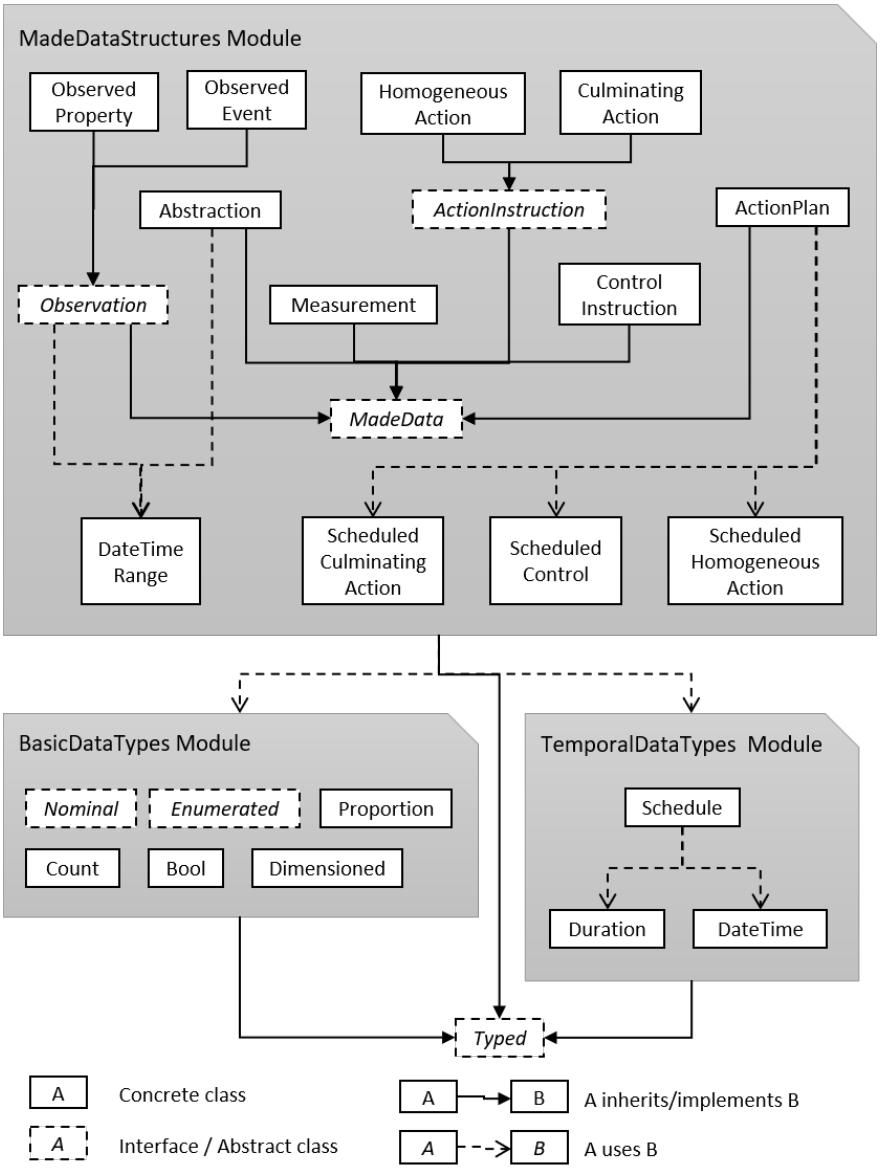

Fig. 5.1 Simplified class diagram showing all the data structures implemented in Rosette for the MADE RIM.

Although not shown in the figure, there is generally a direct correspondence between the components of a data type in the MADE RIM and the fields of the implemented structure. For example, recall that a scheduled homogeneous action is modelled as follows (Eq. 4.31):

$$\begin{gathered} \mathit{ScheduledHomogeneousAction} = \\ \mathit{ActionType} \times \mathit{Schedule} \times \mathit{Rate} \times \mathit{Duration} \end{gathered}$$

This translates to the following in Rosette:

```
(struct scheduled-homogeneous-action
  (action-type schedule rate duration) ...)
```

The `struct` keyword declares a new structure type, in this case with the identifier `scheduled-homogeneous-action`, and it contains four fields as expected: `action-type`, `schedule`, `rate` and `duration`. The ellipses at the end is where the appropriate interfaces are implemented; for scheduled homogeneous actions, this means, for example, implementing `valid?` to ensure that the rate field contains a dimensioned value, the duration field a duration, etc.

The six main data types in the MADE RIM (namely, measurement, observation, abstraction, action plan, action instruction and control instruction) are implemented similarly as Rosette structures, but not all components were translated into fields. More specifically, since every instance of a structure remembers its type (which can be retrieved using `get-type`), it is not necessary for MADE data items to contain a separate ID field to specify its type. In fact, as shown in Fig. 5.1, the primitive datatype ID is not implemented at all; where an explicit identifier is necessary, such as for target processes in control instructions, a symbol is used in place of the ID.

Furthermore, since the main purpose of the MADE archetype (and guideline) language is to capture the clinical knowledge in guidelines, the reference implementation assumes that all MADE data are generated on time. As a result, transaction date-times are not implemented as a field but as a method to ensure that they are equal to the valid date-times of the MADE data (valid start date-times for observed events and abstractions or start date-times for action instructions). For example, recall that a

measurement is modelled as follows (Eq. 4.1):

$$Measurement = MeasurementType \times TransactionDateTime \times ValidDateTime \times Dimensioned$$

In Rosette, the measurement datatype is implemented as follows:

```
1  (struct measurement made-data (valid-datetime value)
2    #:transparent
3    #:methods gen:transaction
4    [(define (transaction-datetime self)
5      (measurement-valid-datetime self))]
6    #:methods gen:typed
7    [(define/generic super-valid? valid?)
8     (define (get-type self) measurement)
9     (define (valid? self)
10      (and (datetime? (measurement-valid-datetime self))
11           (dimensioned? (measurement-value self))
12           (super-valid? (measurement-valid-datetime self))
13           (super-valid? (measurement-value self))
14           (super-valid? (made-data (made-data-proxy-flag self)))
15      ))])
```

It is outside the scope of this thesis to explain the full details of the syntax, but to summarise, the first line shows that the `measurement` structure inherits from `made-data`, which contains a field for the proxy flag, and it extends `made-data` with two additional fields, one for its valid date-time and the other for its measurement value. The measurement ID is not implemented, while the transaction date-time is implemented as the method `transaction-datetime`, which returns the valid-date-time of the measurement (lines 4 and 5). Furthermore, as expected, the `get-type` method (line 8) returns `measurement`, while the `valid?` method (line 9) checks, amongst other things, that `valid-datetime` contains a date-time stamp (line 10) and `value` a dimensioned value (line 11).

5.3.2 Implementation of the Syntactic Forms

The data structures presented above provide the groundwork on which macros are implemented which transform the syntactic forms of the archetype language into the appropriate Rosette constructs. The implementation details are available at https://github.com/nlsfung/MADE-Language/tree/master/lang; to summarise, each archetype is transformed into a structure that inherits from the appropriate MADE data type while re-implementing the `typed` interface. More specifically, for every archetype structure, `get-type` returns its identifier and `valid?` checks whether its instances satisfy its specification. For example, consider the archetype for body temperature, which is reproduced below:

```
(define-measurement body-temperature 'celsius
    (lambda (d) (dim>=? d (dimensioned -273 'celsius))))
```

This is expanded into the following source code in Rosette:

```
1  (struct body-temperature measurement ()
2    #:transparent
3    #:methods gen:typed
4    [(define/generic super-valid? valid?)
5     (define (get-type self) body-temperature)
6     (define (valid? self)
7       (and (super-valid?
8              (measurement (made-data-proxy-flag self)
9                (measurement-valid-datetime self)
10               (measurement-value self)))
11           (eq? (dimensioned-units (measurement-value self))
12                'celsius)
13           ((lambda (d) (dim>=? d (dimensioned -273 'celsius)))
14               (measurement-value self))))])
```

As expected, `body-temperature` inherits from `measurement` (line 1), the `get-type` procedure returns `body-temperature` instead of `measurement` (line 5) and the `valid?` procedure checks whether the unit of measurement is `'celsius`

and whether the measurement value is larger than or equal to 273 (lines 11 to 14). The `super-valid?` procedure (lines 4 and 7) ensures that instances of body temperature also satisfy the specification of measurements, for example, whether its valid date-time is indeed a date-time stamp and whether its value is dimensioned.

Apart from the data structure, the implemented macros also define for each archetype a getter that, as explained before, returns a symbolic instance of that archetype. In Rosette, all symbolic instances must be constructed from symbolic integers, Booleans, bit-vectors and uninterpreted functions, the latter two of which are not applicable for the reference implementation. In other words, to create symbolic instances of an archetype, the getter must instantiate the corresponding data structure and populate the fields with either symbolic or concrete values as appropriate.

In the case of the body temperature archetype, executing the getter `get-body-temperature` without any arguments returns a symbolic body temperature with a variable hour (`dt-part$0`) in its date-time stamp and a variable temperature value (`dim$0`), but the unit is fixed to `'celsius`:

```
> (get-body-temperature)
(body-temperature #f (datetime 2019 12 15 dt-part$0 0 0)
  (dimensioned dim$0 'celsius))
```

Such getters are used by the `verify-archetype` procedure to check for inconsistencies in the archetype specification. Given the archetype identifier (e.g. `id`), executing `verify-archetype` is, in effect, equivalent to executing the following code in Rosette:

```
(solve (assert (valid? (get-id))))
```

`solve` is a procedure provided by Rosette that attempts to find the appropriate values for all symbolic integers, Booleans, bit-vectors and uninterpreted functions such that the input assertions are satisfied. For `verify-archetype`, the assertion is that `get-id` returns a valid instance of archetype `id`. If `(unsat)`is returned, this means that the archetype specification contains inconsistencies.

5.4 Verification of Implementation

The reference implementation was verified using both informal testing and formal verification. As summarised in Table 5.2, arbitrary archetypes were manually specified such that each data structure in the implementation would be instantiated at least once. For each of these archetypes, `verify-archetype` was executed to ensure that its specification is satisfiable, and the resulting solutions were manually checked against the specification of the MADE RIM. Where possible, inconsistencies were also purposefully introduced to the archetype specifications to ensure that they are properly detected by `verify-archetype`. As expected, all tests passed, the details for which are available at https://github.com/nlsfung/MADE-Language/blob/master/exp/TestArchetypes.rkt and also included in Appendix A.

5.5 Discussion

5.5.1 Expressiveness of the MADE Archetype Language

Discounting the use of optional lambda expressions, which allow for arbitrarily complex specifications, the MADE archetype language is very simple. For example, it is not possible to specify that an action plan must contain certain combinations of scheduled instructions, or that certain combinations are mutually exclusive. Similarly, control instruction archetypes cannot constrain the possible schedules and statuses of its instances. In fact, a simple inspection of openEHR's Archetype Description Language (ADL) reveals that it can support constraints not only on individual attributes (i.e. fields) but also on other constraints or groups of constraints as well [81], thus highlighting the inadequacies of the MADE archetype language.

However, the main purpose of this thesis is not to develop a comprehensive archetype language but to enable the representation of clinical guideline knowledge for guideline-based pervasive healthcare systems and to support its formal verification and validation. In this respect, it is crucial that the MADE RIM and the MADE archetype language are not overly restrictive. Once they have been fully verified and validated, further research can be conducted to develop a more powerful archetype

Table 5.2 Summary of the tests conducted for each implemented data structure.

Archetype ID	Tested Data Structure																					Satisfiable?
	Measurement	ObservedProperty	ObservedEvent	Abstraction	ActionPlan	HomogeneousActi	CulminatingAction	ControlInstruction	DateTimeRange	SchedHomogen.	SchedCulmin.	ScheduledControl	DateTime	Schedule	Duration	Nominal	Enumerated	Proportion	Count	Bool	Dimensioned	
1	X												X								X	Y
2	X												X								X	N
3		X											X			X						Y
4		X											X						X			Y
5		X											X						X			N
6			X						X				X							X		Y
7				X					X				X				X					Y
8				X					X				X					X				Y
9				X					X				X					X				N
10					X					X			X	X	X						X	Y
11					X						X		X	X	X				X			Y
12					X						X	X	X	X	X				X			Y
13						X							X		X						X	Y
14						X							X		X						X	N
15							X						X						X			Y
16							X						X							X		N
17								X					X	X	X							Y

language. In fact, the openEHR ADL is independent of the openEHR RIM and may therefore be re-implemented for the MADE RIM.

5.5.2 Utility of the Solver-Aided Syntactic Forms

Given the simplicity of the MADE archetype language, it is clear that the utility of `verify-archetype` is also relatively limited. In fact, unless lambda expressions are used, it is trivial to create an instance of an archetype that satisfies its specification. For example, consider a control instruction archetype, which always exhibit the following form:

```
(define-control-instruction id 'target-1 'target-2 ...)
```

Where `id` and `'target-1`, `'target-2`, etc. are placeholders for the actual archetype name and target processes. A valid instance of this archetype would simply be:

```
(id #f (datetime 2019 12 15 0 0 0) 'target-1 (void) #f)
```

However, as the archetype language becomes more complex in the future, it is expected that Rosette's solver-aided facilities will also become increasingly useful. In fact, by creating symbolic instances of the archetypes using the getters, Rosette may also be used to analyse clinical guidelines, which may in turn help identify issues while formalising them.

Chapter 6

The Guideline Model

6.1 Introduction

The previous two chapters focus on formalising the data flow in the MADE PIM, deriving from it a reference information model as well as an archetype language. In this chapter, the MADE guideline model is specified, which is the counterpart of the MADE reference information model but for processes instead of data. This chapter starts with the related work in Sec. 6.2, wherein the features of existing guideline representation languages are summarised. Although it has already been established in Sec. 3.2 that these existing languages focus on control flow instead of data flow, it is nevertheless useful to review how individual tasks are modelled. This in turn guides the detailed specification of the MADE processes, which is presented in full in Sec. 6.3. As with the MADE RIM, an application of the MADE guideline model is demonstrated in Sec. 6.4 by means of the GDM example presented in Sec. 3.6, while its validation using the complete GDM guideline is reserved for Ch. 8. The chapter concludes with a discussion in Sec. 6.5.

6.2 Related Work

In 2003, Peleg et al. identified eight different dimensions for comparing guideline representation languages [58]: 1) overall organisation of guideline plans, 2) specification of guideline intentions, 3) models of medical actions, 4) decision models,

5) expression languages, 6) data abstractions, 7) models of medical concepts, and 8) models of patient information. Of particular relevance are the specification of guideline intentions, decision models and expression languages as they directly relate to the processing of clinical data. The remaining five dimensions mainly relate to the representation of the data and are therefore outside the scope of this chapter.

In general, decision models can be divided into two categories [66]: decision trees and decision tables. Although interchangeable, decision trees are more suitable in cases where the evaluation of one decision criteria can affect the applicability of the remaining criteria, while decision tables are best for modelling decision criteria that should be evaluated independently of each other. As a result, to simplify the process of formalising clinical guidelines, guideline representation languages typically support combinations of decision models; examples include GLIF3 [7], PROforma [76] and Prodigy [36]. Indeed, since guideline-based systems are traditionally designed to provide decision support to clinicians, existing guideline representation languages generally also support user confirmation and ranking of alternatives [58]. Furthermore, some languages, e.g. ASBRU [65, 68], support the specification of guideline goals and intentions to control the execution of a plan. For example, if a goal is achieved, then the plan being executed may be terminated automatically or recommended for termination.

To specify the decision criteria and guideline goals, guideline languages typically rely on the use of expression languages that can support various operations on patient data items [58], including arithmetic, comparisons, first-order logic, temporal logic and conditionals. For example, GLIF3 relies on the GELLO expression language, which is standardised by HL7 and is based on the general-purpose Object Constraint Language (OCL) for specifying constraints and conditions on UML models [15]. Thus like OCL, GELLO is a declarative, object-oriented language but is specifically adapted for clinical decision support.

Other guideline representation languages rely on similar expression languages, but they all assume the existence of all the required patient data items. To query abstractions that must be generated from available data, Shahar et al. in 2004 proposed the DeGeL framework [69] that integrates, amongst other things, a sub-system (Spock [92]) for executing guidelines and a sub-system (IDAN [5]) for performing knowledge-based temporal abstractions. Such separation of procedural and declara-

tive reasoning tasks is also exhibited by the MADE PIM, but unlike typical guideline representation languages, a key aim of the MADE guideline model is to decompose the guideline knowledge into data-driven, parallel processes.

6.3 The MADE Process Types

6.3.1 Generic MADE Process

Recall from Eq. 3.1 that a data flow process is modelled to comprise four components: an ID (*Id*), a data state (*DataState*), a control state (*ControlState*) and an instructions specification (*InstSpec*). As implied in Fig. 3.2, the MADE guideline model extends this generic data flow model by introducing four sub-types of data flow processes, each of which processes patient data differently and therefore exhibits a different instructions specification: *MSpec* for Monitoring, *ASpec* for Analysis, *DSpec* for Decision and *ESpec* for Effectuation processes:

$$InstSpec = MSpec \cup ASpec \cup DSpec \cup ESpec \tag{6.1}$$

Despite these differences, their basic behaviour remains unchanged as governed by the invariants 3.1 to 3.8. In fact, it is assumed for MADE processes that a data state simply comprises the set of all data previously input into the process. Although the data state should in practice be optimised to reduce memory consumption and computational costs, such as by removing irrelevant data, these are non-functional requirements and are therefore outside the scope of this thesis. The main requirement of the data state is that when a process executes at a given date-time, the result, as governed by *updateDataState*, *updateControlState* and *generateData* in Inv. 3.1, is unaffected by whether the data is newly input into the process or is pre-existent in its data state.

More formally, the data state of a process is defined as follows and satisfies the following invariants:

$$DataState = \mathscr{P}(Data) \tag{6.2}$$

Invariant 6.1. Let s_{ctrl} be an arbitrary control state, s_{inst} an instruction specification, t a date-time stamp. Furthermore, let s_{data} be an arbitrary data state and d_{in} a data set. Then:

$$updateDataState(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = s_{data} \cup d_{in}$$

Invariant 6.2. Let i be an arbitrary process ID, s_{ctrl} an arbitrary control state, s_{inst} an instruction specification, t a date-time stamp. Furthermore, let $s_{data1},\ s_{data2}$ be two arbitrary data states and $d_{in1},\ d_{in2}$ two data sets. Then:

$$s_{data1} \cup d_{in1} = s_{data2} \cup d_{in2} \Rightarrow$$
$$[updateControlState(i, s_{data1}, s_{ctrl}, d_{in1}, t) = updateControlState(i, s_{data2}, s_{ctrl}, d_{in2}, t)$$
$$\wedge\ generateData(s_{data1}, s_{ctrl}, s_{inst}, d_{in1}, t) = generateData(s_{data2}, s_{ctrl}, s_{inst}, d_{in2}, t)]$$

Furthermore, recall that control instructions can only re-schedule and/or pause or resume a process as specified by Eq. 4.22, thus like these control instructions, the control state of a MADE process is modelled to comprise a schedule, which in turn consists of a repeat pattern and repeat interval (Eq. 4.24) as well as a status, which can either be paused or running (Eq. 4.27). Thus a process is activated (i.e. *isProcessActivated* returns true) if and only if the process is running and the current date-time is on its schedule. This leads to the following definition and invariant:

$$ControlState = Schedule \times Status \tag{6.3}$$

Invariant 6.3. Let $s_{ctrl} = ((r_{pat}, r_{int}), s_{stat})$ be an arbitrary control state and t an arbitrary date-time stamp. Then:

$$isProcessActivated(s_{ctrl}, t) \Leftrightarrow [s_{stat} = \text{RUNNING} \wedge onSchedule((r_{pat}, r_{int}), t)]$$

As implied by the semantics of `schedule-instance` and `on-schedule?` in the MADE archetype language (Sec. 5.2.2), a date-time stamp is considered on a schedule if it overlaps with a date-time stamp in the repeat pattern, or it occurs on or after any date-time stamp in the repeat pattern (if the repeat interval is ALWAYS), or it occurs an integral number of repeat intervals after a date-time stamp in the repeat pattern (if the repeat interval is a duration). Thus, *onSchedule* has the following signature and satisfies the following invariant:

$$onSchedule : Schedule \times DateTime \rightarrow Boolean \tag{6.4}$$

Invariant 6.4. Let $s = (r_{pat}, r_{int})$ be an arbitrary schedule and t a date-time stamp. Then:

$$\begin{gathered} onSchedule(s,t) = [(r_{int} = \text{NEVER} \Rightarrow \exists x \in r_{pat}.\ t = x) \wedge \\ (r_{int} = \text{ALWAYS} \Rightarrow \exists x \in r_{pat}.\ t \geq x) \wedge \\ (r_{int} \in Duration \Rightarrow \exists x \in r_{pat},\ n \in \mathbb{N}.\ t = n \cdot r_{int} + x)] \end{gathered}$$

When updating the control state of a process, *updateControlState* checks whether there is a control instruction in its data state or input data that is targeted at the process and is valid at the input date-time. If yes, then the schedule and status of the process are replaced by those in the control instruction (unless it equals to NULL in which case the corresponding attribute is left unchanged). Thus more formally, *updateControlState* satisfies the following two invariants, which govern, respectively, the updating of the schedule and status of a process:

Invariant 6.5. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary process, with $s_{ctrl} = (s_{sched1}, s_{stat1})$. Furthermore, let d_{in} an input data set, t the current date-time stamp and (s_{sched2}, s_{stat2}) be the output of $updateControlState(i, s_{data}, s_{ctrl}, d_{in}, t)$. Then:

$$\begin{gathered} s_{sched2} \neq s_{sched1} \Leftrightarrow \\ \exists d \in \{d \mid d \in d_{in} \cup s_{data} \wedge d \in ControlInstruction\}. \\ \pi_1(d) = i \wedge \pi_3(d) = t \wedge \pi_4(d) \neq \text{NULL} \wedge \pi_4(d) \neq s_{sched1} \end{gathered}$$

Invariant 6.6. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary process, with $s_{ctrl} = (s_{sched1}, s_{stat1})$. Furthermore, let d_{in} an input data set, t the current date-time stamp and (s_{sched2}, s_{stat2}) be the output of $updateControlState(i, s_{data}, s_{ctrl}, d_{in}, t)$. Then:

$$s_{stat2} \neq s_{stat1} \Leftrightarrow$$
$$\exists d \in \{d \mid d \in d_{in} \cup s_{data} \wedge d \in ControlInstruction\}.$$
$$\pi_1(d) = i \wedge \pi_3(d) = t \wedge \pi_5(d) \neq \text{NULL} \wedge \pi_5(d) \neq s_{stat1}$$

The invariants above specify the behaviour of the functions *updateDataState* and *updateControlState* for all MADE processes. As expected, the functionality of *generateData* depends on the specific type of process, but since a MADE network can comprise an arbitrary number of processes, it is assumed for simplicity that each MADE process can only output at most one data item at each time instant. I.e.:

Invariant 6.7. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary process, d_{in} an input data set and t the current date-time. Then:

$$|generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t)| \leq 1$$

6.3.2 Monitoring Processes (*MSpec*)

Conceptually, Monitoring processes are responsible for making observations about the environment, including both the internal as well as the external environment of the patient. Since a Monitoring process can output either an observed property or an observed event, its specification can be divided into the two corresponding categories, i.e.:

$$MSpec = PropertySpec \cup EventSpec \tag{6.5}$$

For simple observed properties such as blood glucose levels, it is envisaged that Monitoring processes would act as a channel that directly converts a measurement to the corresponding observed property. However, in other cases, it may be necessary to perform digital signal processing on the raw measurements, such as to remove

noise and to detect salient features of the input measurement streams. Therefore, for Monitoring processes that output observed properties, their specification comprises:

- A time window indicating the duration beyond which data is considered irrelevant.
- A mathematical function that accepts a set of measurements as input and returns the appropriate property value for the output observed property.
- An output type identifying the specific type of observed property to output.

More formally, the specification of Monitoring processes for observed properties can be defined as follows:

$$PropertySpec = TimeWindow \times ValueFunction \times OutputType, \text{ where} \tag{6.6}$$

$$TimeWindow = Duration \tag{6.7}$$

$$ValueFunction = \mathscr{P}(Measurement) \rightarrow PropertyValue \tag{6.8}$$

$$OutputType = Id \tag{6.9}$$

Note that the transaction and valid date-times of an observed property is automatically set to be the date-time of execution. Therefore, only the property type and property value need specifying. Whenever a Monitoring process for observed properties is activated, all input data (including those stored in its data state) are filtered using the time window and then fed into the value function, resulting in an output observed property with the specified output type and computed value. Thus, more formally, *generateData* satisfies the following invariant:

Invariant 6.8. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Monitoring process for observed properties, with $s_{inst} = (w, f, o)$. Furthermore, let d_{in} be the input data set and t the current date-time. Then assuming *isProcessActivated* returns true (i.e. assuming the process is activated):

$$generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) =$$
$$\{(o, t, t, f(filterMeasurement(s_{data} \cup d_{in}, t, w)))\}$$

filterMeasurement is a function that filters out any input MADE data items that are either not measurements or not relevant at the current date-time (i.e. its valid date-time lies beyond the current date-time window). In other words, *filterMeasurement* has the following signature and satisfies the following invariant:

$$\begin{aligned} \mathit{filterMeasurement} : \mathscr{P}(\mathit{Data}) \times \mathit{DateTime} \times \mathit{TimeWindow} \\ \rightarrow \mathscr{P}(\mathit{Measurement}) \end{aligned} \tag{6.10}$$

Invariant 6.9. Let d_{in} be an arbitrary data set, t a date-time stamp and w a time window. Then:

$$\begin{gathered} \mathit{filterMeasurement}(d_{in}, t, w) = \\ \{d \mid d \in d_{in} \wedge d \in \mathit{Measurement} \wedge \pi_3(d) \geq (t - w) \wedge \pi_3(d) \leq t\} \end{gathered}$$

Unlike observed properties, observed events can exhibit a start and an end, and they can only be assigned a boolean value to indicate whether they occurred or not. Thus Monitoring processes for observed events are specified to comprise:

- A time window and predicate specifying the conditions that indicate the start of the event.
- A time window and predicate specifying the conditions that indicate the end of the event.
- An output type identifying the specific type of observed event to output.

$$\mathit{EventSpec} = \mathit{EventTrigger} \times \mathit{EventTrigger} \times \mathit{OutputType} \tag{6.11}$$

$$\mathit{EventTrigger} = \mathit{TimeWindow} \times \mathit{TriggerPredicate} \tag{6.12}$$

$$\mathit{TriggerPredicate} = \mathscr{P}(\mathit{Measurement}) \rightarrow \mathit{Boolean} \tag{6.13}$$

As with the time window in *PropertySpec*, the time windows in *EventSpec* are used to filter out irrelevant measurements. The remaining measurements are input into the predicates to determine if the start or end of the event is detected. If the start is detected, the Monitoring process will then search for the most recent date-time at

which the end is detected (which may be before or equal to the current date-time). This time period between the end date-time and start (i.e. current) date-time then becomes the valid date-time range of the output event (with value false to indicate that the event has not occurred). Similarly, if the end is detected at the current date-time, the process will search for the most recent date-time starting from which the event occurred. If neither the start or end is detected, then it is not known whether the event is occurring or finished occurring; as a result, no event will be generated by the process.

More formally, for Monitoring processes that output observed events, the function *generateData* satisfies the following three invariants, which respectively specifies:

- The appropriate ID and transaction date-time of any output observed event.
- The necessary and sufficient conditions for an output event to be generated.
- The appropriate valid start times and end times for false events (i.e. events that did not occur).
- The appropriate valid start times and end times for true events (i.e. events that did occur).

Invariant 6.10. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Monitoring process for observed events, with o being its target output type. Furthermore, let d_{in} be the input data set, t the current date-time and $d = (o_{out}, t_{trans}, (t_{start}, t_{end}), b)$ be an arbitrary observed event. Then:

$$generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{d\} \Rightarrow (o_{out} = o \wedge t_{trans} = t)$$

Invariant 6.11. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Monitoring process for observed events, with $s_{inst} = ((w_{start}, p_{start}), (w_{end}, p_{end}), o)$. Furthermore, let d_{in} be the input data set and t the current date-time. Then assuming $isProcessActivated(s_{ctrl}, t)$ returns true:

$$(generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{\} \Leftrightarrow$$
$$[\neg\exists t_{inner} \in DateTime.\ t_{inner} \leq t \wedge$$
$$p_{end}(filterMeasurement(d_{in} \cup s_{data}, t_{inner}, w_{end})) \wedge$$

$$p_{start}(filterMeasurement(d_{in} \cup s_{data}, t, w_{start}))] \wedge$$
$$[\neg\exists t_{inner} \in DateTime.\ t_{inner} \leq t \wedge$$
$$p_{start}(filterMeasurement(d_{in} \cup s_{data}, t_{inner}, w_{start})) \wedge$$
$$p_{end}(filterMeasurement(d_{in} \cup s_{data}, t, w_{end}))])$$

Invariant 6.12. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Monitoring process for observed events, with $s_{inst} = ((w_{start}, p_{start}), (w_{end}, p_{end}), o)$. Furthermore, let d_{in} be the input data set, t the current date-time and $\{(o, t_{trans}, (t_{start}, t_{end}), \text{FALSE})\}$ be the output of $generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t)$. Then:

$$t_{end} = t \wedge$$
$$p_{start}(filterMeasurement(d_{in} \cup s_{data}, t_{end}, w_{start})) \wedge$$
$$p_{end}(filterMeasurement(d_{in} \cup s_{data}, t_{start}, w_{end})) \wedge$$
$$\forall t_{inner} \in DateTime.\ t_{start} < t_{inner} \leq t_{end} \Rightarrow$$
$$\neg p_{end}(filterMeasurement(d_{in} \cup s_{data}, t_{inner}, w_{end}))$$

Invariant 6.13. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Monitoring process for observed events, with $s_{inst} = ((w_{start}, p_{start}), (w_{end}, p_{end}), o)$. Furthermore, let d_{in} be the input data set, t the current date-time and $\{(o, t_{trans}, (t_{start}, t_{end}), \text{TRUE})\}$ be the output of $generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t)$. Then:

$$t_{end} = t \wedge$$
$$p_{end}(filterMeasurement(d_{in} \cup s_{data}, t_{end}, w_{end})) \wedge$$
$$p_{start}(filterMeasurement(d_{in} \cup s_{data}, t_{start}, w_{start})) \wedge$$
$$\forall t_{inner} \in DateTime.\ t_{start} < t_{inner} \leq t_{end} \Rightarrow$$
$$\neg p_{start}(filterMeasurement(d_{in} \cup s_{data}, t_{inner}, w_{start}))$$

6.3.3 Analysis Processes (*ASpec*)

An Analysis process is the process of making abstractions from low-level concepts (i.e. observations) about the environment. In general, producing an abstraction involves removing irrelevant information from the low-level concepts, which, in the

MADE guideline model, involves a combination of the following two procedures as implied in Sec. 4.3.3:

- Reducing the range of possible values a concept or a combination of concepts can take, such as converting a real-valued blood glucose observation into a grade indicating the degree of hyperglycaemia (i.e. high blood glucose) the patient is experiencing.
- Combining observations such that the final result relates to a patient condition that spans an extended period of time. Examples include determining whether the patient is exhibiting good glycaemic control (i.e. stable blood glucose levels) over a period of 7 days.

Thus similar to the functionality of the IDAN sub-system in DeGeL [5], Analysis processes are designed to produce temporal abstractions, i.e. abstractions on data over an extended period of time. However, unlike IDAN which adopts a knowledge-based temporal abstraction methodology, Analysis processes rely on generic mathematical expressions to perform an abstraction, which is akin to the expressions featured in existing guideline representation languages. In particular, the specification of Analysis processes is modelled to comprise:

- An output type identifying the specifying type of abstraction to output.
- A set of abstraction triplets, each containing:
  - A time window for filtering out expired observations.
  - A predicate on the filtered observations that specifies the condition under which an abstraction should be generated.
  - An abstraction function that accepts a set of observations as input and returns the value of the abstraction (if one is generated).

$$ASpec = OutputType \times \mathscr{P}(TimeWindow \times AbstractionPredicate \times AbstractionFunction), \text{ where} \tag{6.14}$$

$$AbstractionPredicate = \mathscr{P}(Observation) \rightarrow Boolean \tag{6.15}$$

$$AbstractionFunction = \mathscr{P}(Observation) \rightarrow AbstractionValue \tag{6.16}$$

Like observed events, abstractions are valid over a date-time range [21]. However, unlike observed events, abstractions are not generated by detecting start and end conditions. Whenever an Analysis process is activated, it iterates through its set of abstraction triplets, and during each iteration, it filters out all expired observations and checks whether the remaining observations satisfy the abstraction predicate or not. If satisfied, the Analysis process applies the corresponding abstraction function to the filtered observations and generates an abstraction with the appropriate abstraction value. The transaction date-time and valid start date-time of the output abstraction (if any) is equal to the current date-time, while the valid end date-time is equal to the closest date-time in the future after which the abstraction is no longer valid (given the existing data). Thus unlike typical guideline representation languages, which only performs "point" abstractions as necessitated by the decision points in the guideline, Analysis processes in the MADE guideline model are designed to generate "temporally extended" abstractions independently of the time points at which they may be required for decision-making.

As an example, consider an Analysis process for detecting a fever from body temperature observations. Suppose for this example that a fever is defined as two or more body temperatures above 37°C within a period of five hours, and let the body temperature observations be as shown by the crosses in Fig. 6.1. Now, if the Analysis process is executed at 18:00, then there will be three observations (at 14:00, 15:30 and 17:00 respectively) that lie within the 5-hour time window and are above the threshold. Therefore, a fever abstraction will be generated as the conditions for a fever are all met. As shown in Fig. 6.1, the valid start date-time will equal 18:00 (i.e. the time of execution), while the valid end-time will equal 20:30. After 20:30, the fever abstraction will no longer be valid as there will only be one observation (at 17:00) that exceeds the threshold and still lies within the same 5-hour time window.

More formally, for Analysis processes, *generateData* satisfies the following three invariants, which specify respectively that:

- *generateData* returns an abstraction if and only if at least one abstraction predicate is satisfied given the filtered data.
- The abstraction ID equals the output type, while the transaction date-time and valid start date-time equal the current date-time.

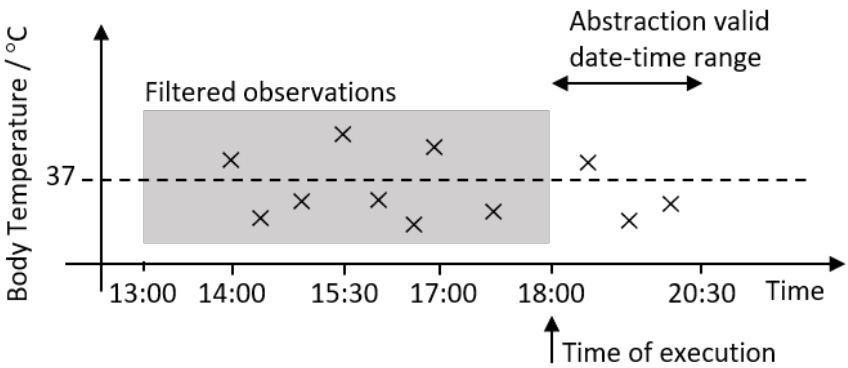

Fig. 6.1 An example to illustrate the valid date-time range of an abstraction. Here, the objective is to detect fever (at 18:00), which is defined as two or more body temperatures above 37°C within a 5-hour window.

- The valid end date-time marks the end of the maximum time interval during which the abstraction is valid given the existing input data.

Invariant 6.14. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Analysis process, with $s_{inst} = (o, \{(w_1, p_1, f_1), (w_2, p_2, f_2), \cdots, (w_N, p_N, f_N)\})$. Furthermore, let d_{in} be the input data set and t the current date-time. Assuming $isProcessActivated(s_{ctrl}, t)$ returns true, then:

$$generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{\} \Leftrightarrow$$
$$[\forall n \in \{1, 2, \cdots, N\}.\ \neg p_n(filterObservation(d_{in} \cup s_{data}, t, w_n))]$$

Invariant 6.15. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Analysis process, d_{in} be the input data set, t the current date-time and $(o_{out}, t_{trans}, (t_{start}, t_{end}), v)$ an arbitrary abstraction. Then:

$$generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{(o_{out}, t_{trans}, (t_{start}, t_{end}), v)\} \Rightarrow$$
$$(o_{out} = o \wedge t_{trans} = t \wedge t_{start} = t)$$

Invariant 6.16. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Analysis process, with $s_{inst} = (o, \{(w_1, p_1, f_1), (w_2, p_2, f_2), \cdots, (w_N, p_N, f_N)\})$. Furthermore, let d_{in} be the

input data set, t the current date-time and $(o, t, (t_{start}, t_{end}), v)$ be the output of invoking $generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t)$. Then:

$$\exists n \in \{1, 2, \cdots, N\}.\ \forall t_{inner} \in DateTime \geq t_{start}.$$
$$t_{inner} \leq t_{end} \Leftrightarrow$$
$$[p_n(filterObservation(d_{in} \cup s_{data}, t_{inner}, w_n)) \wedge$$
$$f_n(filterObservation(d_{in} \cup s_{data}, t_{inner}, w_n)) = v]$$

As expected, *filterObservation* is the counterpart of *filterMeasurement* for filtering out any input MADE data items that are neither observations nor valid within at the current date-time window. For observed properties, an observation is invalid if its valid date-time stamp lies beyond the current date-time window. For observed events, an observation is invalid if its valid date-time range shares no overlap with the current date-time window. Formally, *filterObservation* has the following signature and satisfies the following invariant:

$$filterObservation : \mathscr{P}(Data) \times DateTime \times TimeWindow$$
$$\rightarrow \mathscr{P}(Observation) \tag{6.17}$$

Invariant 6.17. Let d_{in} be an arbitrary data set, t a date-time stamp and w a time window. Then:

$$filterObservation(d_{in}, t, w) =$$
$$\{d \mid d \in d_{in} \wedge d \in ObservedProperty \wedge \pi_3(d) \geq (t - w) \wedge \pi_3(d) \leq t\} \cup$$
$$\{d \mid d \in d_{in} \wedge d \in ObservedEvent \wedge \pi_2(\pi_3(d)) \geq (t - w) \wedge \pi_1(\pi_3(d)) \leq t\}$$

6.3.4 Decision Processes (*DSpec*)

Decision processes decide on the appropriate plan (i.e. course of action) given the current state of the environment as captured by the abstractions produced by Analysis processes. Since Decision processes (and all other MADE processes) are modelled to execute in parallel during the patient's daily life, the MADE guideline model adopts

decision tables to represent decision-making; decision trees can only be simulated by judicious use of multiple Decision and Effectuation processes. In fact, since MADE processes are modelled to only output at most one data item, alternative plans can also only be generated using multiple Decision processes, one for each plan.

More specifically, Decision processes are modelled to comprise:

- A plan template that can be instantiated into an actual action plan for effectuation. It in turn comprises a plan type that identifies the type of action plan to instantiate as well as a set of three different types of instruction templates specifying the control, homogeneous action and culminating action instructions that constitute the plan.
- A set of decision criterion that determines when the plan template should be instantiated. Each criterion is a function that returns true or false given a set of input abstractions.

$$DSpec = PlanTemplate \times \mathscr{P}(DecisionCriterion), \text{ where} \tag{6.18}$$

$$PlanTemplate = PlanType \times \mathscr{P}(ControlTemplate$$
$$\cup\, HomogeneousActionTemplate \cup CulminatingActionTemplate) \tag{6.19}$$

$$DecisionCriterion = \mathscr{P}(Abstraction) \rightarrow Boolean \tag{6.20}$$

Unlike Monitoring and Analysis processes, Decision processes do not require a time window. This is because the only abstractions that are relevant for Decision processes during execution are those that are valid at the current date-time. These abstractions are checked against the decision criteria, and if any criterion is triggered, an action plan will be instantiated from the plan template. The valid date-time of the output action plan is equal to the current date-time, and the schedule of each instruction in the plan is computed from the current date-time and the relative schedule of the corresponding template.

More formally, let *filterAbstraction* be a function for filtering abstractions, which is largely equivalent to the functions *filterObservation* and *filterMeasurement* for filtering observations and measurements respectively. However, since the abstractions

generated by Analysis processes may extend into the future and may therefore be superseded with other, more recent abstractions, *filterAbstraction* removes abstractions based not only on their valid date-time range but also on the presence of other abstractions of the same type but with more recent transaction date-times. Thus, *filterAbstraction* has the following signature and satisfies the following invariant:

$$\mathit{filterAbstraction} : \mathscr{P}(\mathit{Data}) \times \mathit{DateTime} \rightarrow \mathscr{P}(\mathit{Abstraction}) \tag{6.21}$$

Invariant 6.18. Let d_{in} be an arbitrary data set and t a date-time stamp. Then:

$$\begin{gathered}\mathit{filterAbstraction}(d_{in}, t) = \\ \{d \mid d \in d_{in} \wedge d \in \mathit{Abstraction} \wedge \pi_2(\pi_3(d)) \geq t \wedge \pi_1(\pi_3(d)) \leq t \wedge \\ \neg\exists d_{inner} \in d_{in}.\ \pi_1(d_{inner}) = \pi_1(d) \wedge \pi_2(d_{inner}) > \pi_2(d) \wedge \\ \pi_2(\pi_3(d_{inner})) \geq t \wedge \pi_1(\pi_3(d_{inner})) \leq t\}\end{gathered}$$

Furthermore, let *instantiatePlan* be a function for instantiating the given plan template at the given date-time. It has the following signature and satisfies the following invariant, which ensures that:

- The instantiated plan has the same type as the input plan template.
- The transaction and valid date-time of the action plan are set to the given date-time.
- All scheduled instruction templates are instantiated accordingly in the output action plan.

$$\mathit{instantiatePlan} : \mathit{PlanTemplate} \times \mathit{DateTime} \rightarrow \mathit{ActionPlan} \tag{6.22}$$

Invariant 6.19. Let p_{temp} be an arbitrary plan template and t a date-time stamp. Then:

$$\begin{gathered}\mathit{instantiatePlan}(p_{temp}, t) = (\pi_1(p_{temp}), t, t, \\ \{\mathit{instantiateScheduledInstruction}(i, t) \mid i \in \pi_2(p_{temp})\})\end{gathered}$$

Instantiating an instruction template involves, amongst other things, converting its relative schedule (if present) into a concrete schedule. In the MADE guideline model, a relative schedule comprises:

- A duration indicating the rounding factor with which to round the input date-time.
- A list of durations which specify the starting pattern for the schedule.
- A repeat interval, which may be a duration or the values ALWAYS or NEVER as specified in Eq. 4.26.

$$RelativeSchedule = Duration \times \mathscr{P}(Duration) \times RepeatInterval \tag{6.23}$$

For example, a relative schedule for measuring blood glucose levels might indicate that the patient should start the blood glucose measurements on the next day at 7:00, 9:00, 13:00 and 20:00, repeating daily. When instantiated on May 14, 2020, for example, the actual schedule for monitoring blood glucose levels would start on May 15, 2020 at 7:00, 9:00, 13:00 and 20:00, and this schedule would repeat daily thereafter.

More formally, let *instantiateSchedule* be a function for instantiating a relative schedule at the given date-time. It has the following signature and satisfies the following invariant:

$$instantiateSchedule : RelativeSchedule \times DateTime \rightarrow Schedule \tag{6.24}$$

Invariant 6.20. Let r be an arbitrary relative schedule and t a date-time stamp. Then:

$$instantiateSchedule(r,t) =$$
$$(\{p + round(t, \pi_1(r)) \mid p \in \pi_2(r)\}, \pi_3(RelativeSchedule))$$

Apart from a relative schedule, instruction templates also contain the necessary elements to populate the corresponding elements in the instantiated scheduled instructions. For example, templates for scheduled homogeneous actions (i.e.

HomogeneousActionTemplate) also contain an action type, rate and duration, thus they are formally defined as follows, with *instantiateScheduledInstruction* satisfying the following invariant for scheduled homogeneous actions:

$$\begin{gathered} HomogeneousActionTemplate = \\ ActionType \times RelativeSchedule \times Rate \times Duration \end{gathered} \tag{6.25}$$

Invariant 6.21. Let i be a template for an arbitrary scheduled homogeneous action, and let t be an arbitrary date-time stamp. Then:

$$\begin{gathered} instantiateScheduledInstruction(i,t) = \\ (\pi_1(i), instantiateSchedule(\pi_2(i),t), \pi_3(i), \pi_4(i)) \end{gathered}$$

For scheduled culminating actions, their templates comprise an action type, a relative schedule and a goal state. I.e.:

$$\begin{gathered} CulminatingActionTemplate = \\ ActionType \times RelativeSchedule \times GoalState \end{gathered} \tag{6.26}$$

Invariant 6.22. Let i be a template for an arbitrary scheduled culminating action, and let t be an arbitrary date-time stamp. Then:

$$\begin{gathered} instantiateScheduledInstruction(i,t) = \\ (\pi_1(i), instantiateSchedule(\pi_2(i),t), \pi_3(i)) \end{gathered}$$

For scheduled control instructions, their templates comprise a target process, an optional relative schedule and an optional process status. If a relative schedule is not provided, then the instantiated scheduled control instruction should also not contain a schedule. This leads to the following definition and invariant:

$$\begin{gathered} ControlTemplate = \\ TargetProcess \times (RelativeSchedule \cup \{\text{NULL}\}) \times (Status \cup \{\text{NULL}\}) \end{gathered} \tag{6.27}$$

Invariant 6.23. Let i be a template for an arbitrary scheduled control instruction, and let t be an arbitrary date-time stamp. Then:

$$[\pi_2(i) = \text{NULL} \Rightarrow \mathit{instantiateScheduledInstruction}(i,t) = (\pi_1(i), \text{NULL}, \pi_3(i))] \wedge$$
$$[\pi_2(i) \in \mathit{RelativeSchedule} \Rightarrow \mathit{instantiateScheduledInstruction}(i,t) = (\pi_1(i), \mathit{instantiateSchedule}(\pi_2(i),t), \pi_3(i))]$$

The invariants thus far, namely Inv. 6.19 to 6.23, specify how an action plan is to be instantiated from a plan template. A Decision process instantiates an action plan if and only if the filtered abstractions satisfy one or more of its decision criterion. This leads to the following two invariants, which specify respectively that:

- An action plan is generated if an only if at least one decision criterion is satisfied.
- The output action plan (if any) is an instantiation of the plan template at the current date-time.

Invariant 6.24. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Decision process, with $s_{inst} = (p_{temp}, \{c_1, c_2, \cdots, c_N\})$. Furthermore, let d_{in} be the input data set and t the current date-time. Assuming $\mathit{isProcessActivated}(s_{ctrl}, t)$ returns true, then:

$$\mathit{generateData}(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{\} \Leftrightarrow$$
$$[\forall n \in \{1, 2, \cdots, N\}.\ \neg c_n(\mathit{filterAbstraction}(d_{in} \cup s_{data}, t))]$$

Invariant 6.25. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Decision process, d_{in} be the input data set, t the current date-time and d_{out} an arbitrary action plan. Then:

$$\mathit{generateData}(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{d_{out}\} \Rightarrow$$
$$d_{out} = \mathit{instantiatePlan}(\pi_1(s_{inst}), t)$$

6.3.5 Effectuation Processes (*ESpec*)

Conceptually, effectuation involves performing the decided the course of action, with the intention of bringing about a patient's state (whether directly or via changes in the patient's external environment). In the MADE guideline model, Effectuation processes comprise:

- A specification of the target scheduled instructions to effectuate. These targets are identified by its type of action plan, its type of target instruction (which may be an action type for action instructions or a target process for control instructions) as well as a predicate that must be satisfied by the effectuated instruction.
- An output type identifying the specific type of instruction to output.

$$ESpec = \mathscr{P}(TargetScheduledInstruction) \times OutputType \tag{6.28}$$

$$TargetScheduledInstruction = PlanType \times (ActionType \cup TargetProcess) \times InstructionPredicate \tag{6.29}$$

$$InstructionPredicate = (ScheduledControl \cup ScheduledHomogeneousAction \cup ScheduledCulminatingAction) \rightarrow Boolean \tag{6.30}$$

When activated, an Effectuation process filters out any action plans that are not valid at the current date-time and extracts, from the remaining action plans, all relevant scheduled instructions based on the specified targets. From these remaining scheduled instructions, the Effectuation process then determines if any should be effectuated. Scheduled control instructions are effectuated immediately (if they match the targets) while scheduled action instructions are only effectuated if the current date-time matches their schedule.

Note that since all scheduled instructions already specify their action type (for action instructions) or target process (for control instructions), it is not necessary for the Effectuation process to also specify an output type. However, by keeping the target input and output types separate, the scheduled instructions can be specified

at a different level of abstraction than the effectuated instruction. For example, the schedule may simply indicate that the patient should exercise for 3 hours, while the effectuated instruction may specify explicitly that the patient should run on a treadmill.

More formally, *generateData* for Effectuation processes satisfies the following four invariants. The first invariant indicates that an instruction can be generated only if a scheduled instruction matches one of the specified targets. The remaining three invariants specify respectively that the contents of the output homogeneous action instruction, culminating action instruction and control instruction must match those of the corresponding scheduled instruction.

Invariant 6.26. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Effectuation process, with $s_{inst} = (\{x_1, x_2, \cdots, x_N\}, o)$. Furthermore, let d_{in} be the input data set and t the current date-time. Assuming $isProcessActivated(s_{ctrl}, t)$ returns true, then:

$$\begin{gathered}[\forall n \in \{1, 2, \cdots, N\}.\ \forall a \in filterPlan(s_{data} \cup d_{in}, t).\ \forall y \in \pi_4(a). \\ \pi_1(x_n) \neq \pi_1(a) \vee \pi_2(x_n) \neq \pi_1(y) \vee \neg\pi_3(x_n)(y)] \Rightarrow \\ generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{\}\end{gathered}$$

Invariant 6.27. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Effectuation process, with $s_{inst} = (\{x_1, x_2, \cdots, x_N\}, o)$. Furthermore, let d_{in} be the input data set, t the current date-time and d_{out} an arbitrary MADE data item. Then:

$$\begin{gathered}[generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{d_{out}\} \wedge \\ d_{out} \in HomogeneousAction] \Rightarrow \\ [\pi_1(d_{out}) = o \wedge \pi_2(d_{out}) = t \wedge \pi_3(d_{out}) = t \wedge \\ \exists n \in \{1, 2, \cdots, N\}.\ \exists a \in filterPlan(s_{data} \cup d_{in}, t).\ \exists y \in \pi_4(a). \\ \pi_1(x_n) = \pi_1(a) \wedge \pi_2(x_n) = \pi_1(y) \wedge \pi_3(x_n)(y) \wedge \\ onSchedule(\pi_2(y), t) \wedge \pi_4(d_{out}) = \pi_3(y) \wedge \pi_5(d_{out}) = \pi_4(y)]\end{gathered}$$

Invariant 6.28. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Effectuation process, with $s_{inst} = (\{x_1, x_2, \cdots, x_N\}, o)$. Furthermore, let d_{in} be the input data set, t the current

date-time and d_{out} an arbitrary MADE data item. Then:

$$[generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{d_{out}\} \wedge$$
$$d_{out} \in CulminatingAction] \Rightarrow$$
$$[\pi_1(d_{out}) = o \wedge \pi_2(d_{out}) = t \wedge \pi_3(d_{out}) = t \wedge$$
$$\exists n \in \{1, 2, \cdots, N\}.\ \exists a \in filterPlan(s_{data} \cup d_{in}, t).\ \exists y \in \pi_4(a).$$
$$\pi_1(x_n) = \pi_1(a) \wedge \pi_2(x_n) = \pi_1(y) \wedge \pi_3(x_n)(y) \wedge$$
$$onSchedule(\pi_2(y), t) \wedge \pi_4(d_{out}) = \pi_3(y)]$$

Invariant 6.29. Let $p = (i, s_{data}, s_{ctrl}, s_{inst})$ be an arbitrary Effectuation process, with $s_{inst} = (\{x_1, x_2, \cdots, x_N\}, o)$. Furthermore, let d_{in} be the input data set, t the current date-time and d_{out} an arbitrary MADE data item. Then:

$$[generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{d_{out}\} \wedge$$
$$d_{out} \in ControlInstruction] \Rightarrow$$
$$[\pi_1(d_{out}) = o \wedge \pi_2(d_{out}) = t \wedge \pi_3(d_{out}) = t \wedge$$
$$\exists n \in \{1, 2, \cdots, N\}.\ \exists a \in filterPlan(s_{data} \cup d_{in}, t).\ \exists y \in \pi_4(a).$$
$$\pi_1(x_n) = \pi_1(a) \wedge \pi_2(x_n) = \pi_1(y) \wedge \pi_3(x_n)(y) \wedge$$
$$\pi_4(d_{out}) = \pi_2(y) \wedge \pi_5(d_{out}) = \pi_3(y)]$$

As expected, *filterPlan* is a function for filtering out MADE data items that are either not action plans or are irrelevant at the current date-time. However, unlike measurements, observations and abstractions, an action plan is considered relevant if (and only if) there does not exist another action plan with the same plan type and a more recent valid date-time. This ensures that the instructions of an action plan will continually be effectuated until the plan itself is replaced. More formally,

$$filterPlan : \mathscr{P}(Data) \times DateTime \rightarrow \mathscr{P}(ActionPlan) \tag{6.31}$$

Invariant 6.30. Let d_{in} be an arbitrary data set and t a date-time stamp. Then:

$$filterPlan(d_{in}, t) = \{d \mid d \in d_{in} \wedge d \in ActionPlan \wedge \neg\exists d_{inner} \in d_{in}.$$

$$d_{inner} \in ActionPlan \wedge \pi_1(d_{inner}) = \pi_1(d) \wedge \pi_3(d_{inner}) > \pi_3(d) \wedge \pi_3(d_{inner}) \leq t\}$$

6.4 Application Example

As an example application of the MADE guideline model, consider the clinical guideline fragment that was introduced in Sec. 3.6 and for which a MADE domain information model was derived in Sec. 4.4. As shown in Fig. 3.5, this fragment comprises three processes, namely an Analysis process (*AnalyseKetonuria*) for analysing the presence of ketonuria, a Decision process (*DecideIncreaseCarbohydrates*) for deciding to increase the patient's carbohydrates intake, as well as an Effectuation process (*EffectuateCarbohydratesIntake*) for effectuating the increase in carbohydrates intake.

Based on the guideline fragment (Fig. 3.4) and the corresponding MADE PIM, the Analysis process (*AnalyseKetonuria*) can be formalised as follows:

$$\begin{gathered} AnalyseKetonuria \subset Analysis, \text{ such that} \\ \forall p \in AnalyseKetonuria.\ \pi_1(p) = \textsc{Analyse Ketonuria} \wedge \\ \pi_1(\pi_4(p)) = \textsc{Ketonuria} \wedge \pi_2(\pi_4(p)) = \{(\textsc{One Week}, \\ [\lambda d_{in}.\ |\{d \mid d \in d_{in} \wedge d \in UrinaryKetoneLevel \wedge \pi_4(d) \in \{+,++\}\}| \geq 3], \\ [\lambda d_{in}.\ \textsc{Positive}])\} \end{gathered} \tag{6.32}$$

As expected, *AnalyseKetonuria* outputs ketonuria (KETONURIA) abstractions, and it operates on data within a 7-day time window. For this process, the condition for generating an abstraction is the presence of three or more "positive" urinary ketone levels (+ or + +) in the filtered data, and the resulting value for the ketonuria abstraction is "positive" (POSITIVE).

Similarly, the Decision process (*DecideIncreaseCarbohydrates*) is formalised as follows. This process only has one decision criterion ($\pi_2(\pi_4(p))$), namely to check whether ketonuria is positive and whether the patient is compliant with her diet. If satisfied, the Decision process then instantiates the dietary plan (DIETARY PLAN) to increase carbohydrates intake by 10 g. The exact schedule for when to increase carbohydrates intake is not specified by the guideline fragment (except that it should

be at dinner or at bedtime), so for this example, it is assumed that carbohydrates intake should be increased starting from the following day at 19:00 (repeating daily).

$$\begin{gathered}
\textit{DecideIncreaseCarbohydrates} \subset \textit{Decision}, \text{ such that} \\
\forall p \in \textit{DecideIncreaseCarbohydrates}. \\
\pi_1(p) = \textsc{Decide Increase Carbohydrates} \wedge \\
\pi_1(\pi_1(\pi_4(p))) = \textsc{Dietary Plan} \wedge \\
\pi_2(\pi_1(\pi_4(p))) = \{(\textsc{Change Diet Instruction}, \\
(\textsc{One Day}, \{19{:}00\}, \textsc{One Day}), 10\,\textsc{g})\} \wedge \\
\pi_2(\pi_4(p)) = \{[\lambda d_{in}. (\exists d \in d_{in}.\ d \in \textit{Ketonuria} \wedge \pi_4(d) = \textsc{Positive}) \wedge \\
(\exists d \in d_{in}.\ d \in \textit{DietCompliance} \wedge \pi_4(d) = \textsc{Compliant})]\}
\end{gathered} \tag{6.33}$$

Finally, the Effectuation process (*EffectuateCarbohydratesIntake*) can be formalised as follows. It searches for dietary plans (DIETARY PLAN) and effectuates the instructions for adjusting the patient's diet (CHANGE DIET INSTRUCTION). Note that this process is a proxy process as the instruction (to increase carbohydrates intake) must be performed manually by the patient herself. As a result, the output instruction type (CHANGE DIET PROXY) is a proxy version of CHANGE DIET INSTRUCTION.

$$\begin{gathered}
\textit{EffectuateCarbohydratesIntake} \subset \textit{Effectuation}, \text{ such that} \\
\forall p \in \textit{EffectuateCarbohydratesIntake}. \\
\pi_1(p) = \textsc{Effectuate Carbohydrates Intake} \wedge \\
\pi_1(\pi_4(p)) = \{(\textsc{Dietary Plan}, \textsc{Change Diet Instruction}, \lambda d_{in}.\ \textsc{True})\} \wedge \\
\pi_2(\pi_4(p)) = \textsc{Change Diet Proxy}
\end{gathered} \tag{6.34}$$

6.5 Discussion

6.5.1 Expressiveness of the Guideline Model

In theory, the MADE guideline model is Turing complete as it supports conditional branching as well as the reading and writing of arbitrary amounts of memory. However, each process type is given a specific clinical interpretation, thus in this respect they exhibit limited expressiveness. For example, Decision processes in the MADE guideline model can ultimately be reduced to if-then-else expressions, with the condition being the decision criteria and the result being the instantiated action plan. This is in contrast with the traditional notion of planning in artificial intelligence, which typically involves a complex reasoning procedure for creating a plan from basic plan components [8].

Indeed, compared with typical guideline representation languages, the MADE guideline model provides less intrinsic support for representing tasks that require manual intervention. A clear example is support for alternative plans, which is typically provided by other guideline representation languages. However, the main purpose of the MADE guideline model is to enable pervasive healthcare systems to provide guideline-based support to patients, which it achieves by capturing the data flow instead of control flow in clinical guidelines. In this way, the MADE guideline model intrinsically allows all processes to be distributed and executed in parallel by such systems. Furthermore, given the potential amounts of data that must be processed, it is envisaged in the pervasive healthcare setting that many processes should be automated by the system.

6.5.2 Computability of the Guideline Model

One typical advantage of reduced expressiveness in computer-interpretable languages (including knowledge representation languages) is increased amenability to analysis. Ideally, the MADE processes should be free from errors during execution, especially since they are designed for clinical purposes. Therefore, by keeping the MADE guideline model simple, it is envisaged that the process of verifying its implementation and formalised guidelines will also be facilitated.

In fact, the MADE processes are designed to guarantee termination during execution, provided that the user-provided functions (e.g. predicates and decision criteria) do not contain arbitrary recursions and/or iterations. The processes do not support arbitrary branching, and their behaviour can all be captured using non-recursive functions or primitive recursive functions applied over a finite domain, such as the set of input data.

For MADE networks, there is a clear sequential flow of data from Monitoring to Effectuation processes, and as implied by Inv. 3.5, control instructions cannot affect the control states of the processes in the same time instant that they are generated. As a result, even though MADE networks are designed to generate data until a closure is reached (as specified by Inv. 3.4), this closure is always finite and can always terminate. The only assumption is that the different types of MADE data items do not overlap, such that an culminating action instruction, for example, cannot also be an observed property.

Furthermore, because only data and not control is communicated between processes, concurrency problems such as live locks and dead locks will not occur. Although the MADE models do assume perfect communication between processes (Sec. 3.3.2), and problems may clearly arise if data are in any way delayed, this is a technical concern and is therefore considered outside the scope of the MADE models, which only focus on the clinical concerns of pervasive healthcare.

However, although MADE processes (and MADE networks) may be guaranteed to terminate, their behaviour may not always be deterministic. More specifically, non-deterministic behaviour may be exhibited by Monitoring, Analysis and Effectuation processes under the following respective conditions:

- The conditions indicating the start of an observed event overlap with those indicating the end of the same event for a Monitoring process, such that the process cannot always distinguish whether the event has started or ended.
- The abstraction triplets of an Analysis process are not mutually exclusive, such that multiple triplets may be triggered at the same time, each resulting in an abstraction with potentially a different value and valid date-time range.
- The target scheduled instructions of an Effectuation process are not mutually exclusive, such that multiple instructions may require effectuation at the same

time, even though all processes by definition can output at most one data item at each time instant.

One possible solution to this non-deterministic behaviour is to allow processes to output a set of data items instead of only a single data item. In this way, an Analysis process, for example, can be specified to output all the abstractions corresponding to all the triggered abstraction triplets. However, this approach will not in fact address the root cause of the problem, which is that the specifications of the processes are not mutually exclusive. For example, it is clearly nonsensical for a Monitoring process to return two contradictory observations at the same time to indicate that an event has both started and ended. As a result, it is hypothesised that a more appropriate solution to this non-deterministic behaviour is to provide tool support to detect it during run-time or during the formalisation of clinical guidelines.

Chapter 7

The Guideline Language

7.1 Introduction

In this chapter, the MADE guideline language is presented, which is the counterpart to the MADE archetype language but for the guideline model instead of the reference information model. In particular, this guideline language builds on top of the MADE archetype language such that the tasks specified in clinical guidelines can be formalised into MADE processes. Indeed, since it is also implemented on top of Rosette, the MADE guideline language supports all constructs provided by Rosette as well as those exposed by the implementation of the language (including the MADE RIM and the MADE guideline model). However, this chapter only focuses on the constructs directly relevant for formalising guidelines.

In Sec. 7.2, the complete syntax of the MADE guideline language is presented using the same EBNF notation explained in Sec. 5.1 and adopted in Sec. 5.2. The semantics of the MADE guideline language is also described with examples in Sec. 7.2, and in Sec. 7.3, its reference implementation is explained. To verify the implementation, invariants on the MADE guideline model are checked against the implementation using Rosette's solver-aided features, the details of which are presented in Sec. 7.4. Finally, this chapter concludes with a discussion in Sec. 7.5.

7.2 Language Specification

7.2.1 MADE Data Items

Since MADE processes operate on MADE data items, specifications in the MADE guideline language may involve manipulating and instantiating them. As described in Sec. 5.3.2, archetypes are transformed into structures (that inherit from the appropriate MADE data type), thus MADE data items are instantiated in the same manner as other structures in Rosette. More specifically, the syntax for instantiating a MADE data item is as follows:

```
data-item : '(', ID, EXPR, {EXPR}, ')';
```

`ID` is the identifier of the structure to instantiate, and each expression assigns a value to the corresponding field in the structure; in other words, there must be the same number of expressions as there are fields in the structure (including fields inherited from parent structures).

Furthermore, let `struct-id` be an identifier for a structure and let `field-id` be an identifier for a field in `struct-id`. Then `struct-id-field-id` is the procedure for extracting the value of `field-id` from the given instance of `struct-id`. For example, the following commands:

1. Creates a new structure type with Id `parent-struct` and field `parent-field`.
2. Creates a new structure type with Id `test-struct` that inherits from the structure `parent-struct` and contains an extra field `test-field`.
3. Instantiates `test-struct` with a value of 10 for `parent-field` and 20 for `test-field`.
4. Retrieves the values of `parent-field` and `test-field` in the instantiated `test-struct`.

```
> (struct parent-struct (parent-field))
> (struct test-struct parent-struct (test-field))
> (define test-instance (test-struct 10 20))
> (parent-struct-parent-field test-instance)
10
```

```
> (test-struct-test-field test-instance)
20
```

As expected, the fields of each MADE item varies according to their type, whether they are measurement, observations (for properties or events), abstractions, action plans, action instructions (homogeneous or culminating) or control instructions:

- Measurements (`measurement`) inherit the proxy flag (`proxy-flag`) from the generic MADE data structure (`made-data`) and contain fields for their valid date-time (`valid-datetime`) and value (`value`).
- Observations for properties (`observed-property`) also inherit `proxy-flag` and contain the fields `valid-datetime` and `value`. However, observations for events (`observed-event`) contain a field for a valid date-time range (`valid-datetime-range`) instead of a singular valid date-time.
- Abstractions (`abstraction`), like observed events, inherit `proxy-flag` and contain the fields `valid-datetime-range` and `value`.
- Action plans (`action-plan`), apart from inheriting `proxy-flag`, also contain the fields `valid-datetime` and `instruction-set`; the latter is for the set of scheduled instructions that constitute the plan.
- Homogeneous action instructions (`homogeneous-action`) also inherit the field `proxy-flag`, but they contain the fields `start-datetime`, `rate` and `duration` for their starting date-time, rate and duration respectively. Culminating action instructions (`culminating-action`) contain the field `goal-state` for their goal states instead of a rate and duration.
- Control instructions (`control-instruction`) inherit `proxy-flag` and contain the fields `target-process`, `valid-datetime`, `schedule` and `status` for their target process, valid date-time, target schedule and target status respectively.

For example, the following syntax creates an archetype for an observation about the severity of a burn and instantiates a 3rd degree burn on April 18, 2020 at 9:00 am. Various procedures are used to extract the values of the corresponding fields in the observation:

```
> (define-observation burn-severity
    enumerated '1st '2nd '3rd '4th)
> (define instance
    (burn-severity #f (datetime 2020 4 18 9 0 0)
      (burn-severity-value-space '3rd)))
> (observed-property-valid-datetime instance)
(datetime 2020 4 18 9 0 0)
> (observed-property-value instance)
(burn-severity-value-space '3rd)
> (made-data-proxy-flag instance)
#f
```

Once extracted, the fields can be operated on such as described in Sec. 5.2.1 for basic data types and Sec. 5.2.2 for temporal data types. However, MADE data items can also be compared against each other for equality using the `eq?` procedure, which checks whether the two input MADE data items are of the same type and whether each field contain values that are equal (as determined by `eq?`). Returning to the burn severity example:

```
> (eq? instance
       (burn-severity #f (datetime 2020 4 18 9 0 0)
        (burn-severity-value-space '3rd)))
#t
> (eq? instance
       (burn-severity #f (datetime 2020 4 18 9 0 0)
        (burn-severity-value-space '4th)))
#f
```

Furthermore, as explained in Sec. 5.3.1, the transaction date-time of a MADE data item can be retrieved using the procedure `transaction-datetime`. Other procedures provided by the implementation to manipulate MADE data items include `get-type`, which returns the archetype Id of the MADE data item, and `valid?`, which checks whether the fields contain values of the expected type:

```
> (transaction-datetime instance)
```

```
(datetime 2020 4 18 9 0 0)
> (eq? (get-type instance) burn-severity)
#t
> (valid? instance)
#t
> (valid? (burn-severity #f 100 100))
#f
```

7.2.2 Monitoring Processes

Recall from Sec. 6.3.2 that two types of Monitoring processes are distinguished, those that output observed properties and those that output observed events. For the monitoring of observed properties, the syntactic form comprises the following components to reflect its specification (Eq. 6.6):

- The `#:property` keyword.
- An identifier (ID) for the new process.
- A boolean indicating whether the process is a proxy or not, i.e. whether the output of the process requires manual confirmation or not.
- An identifier indicating the output type of the process. This identifier should correspond to an observed property as specified using the MADE archetype language.
- The time window for the process, which is represented as a duration.
- The value function for the process, which is represented as a lambda expression.

```
monitoring-process-property :
  '(define-monitoring #:property', ID, proxy, output-type,
      window, LAMBDA-EXPR, ')';
proxy : BOOLEAN;
output-type : ID;
window : duration-instance;
```

Note that in the specification of the MADE PIM (Sec. 3.5.2), proxy processes are distinguished from regular, non-proxy processes by their Id. However, in the MADE guideline language, proxy processes are identified via an additional boolean proxy flag. While unnecessary, this allows a clear distinction between the clinical purpose of the process (as captured by its Id), and whether the process requires manual confirmation. Furthermore, it allows the implementation to automatically detect proxy processes and ensure that the data generated by such processes are not operated on by other MADE processes.

The syntactic form for the monitoring of observed events (which is specified in Eq. 6.11) is equivalent to that for observed properties, except that the keyword is `#:event` instead of `#:property`. Furthermore, instead of a time window and value function, the syntactic form contains two trigger pairs, one to specify the conditions indicating the start of the event and the other the end of the event. Each trigger pair contains a time window and a predicate (expressed in the form of a lambda expression).

```
monitoring-process-event :
  '(define-monitoring #:event', ID, proxy, output-type,
      2 * trigger, ')';
trigger : '(event-trigger', window, LAMBDA-EXPR, ')';
```

For example, the following is a contrived Monitoring process for detecting the event that a patient is sprinting. The exercise event is considered to have started if the patient's average speed has reached 3 ms^{-1} in a 5-second window, and it is considered to have stopped if the patient's average speed has dropped below 2 ms^{-1} in a 5-second window.

```
(define-monitoring #:event monitor-sprint #f sprint-event
  (event-trigger
    (duration 0 0 0 5)
    (lambda (dSet)
      (dim>=? (average (filter (lambda (d) (body-speed? d))
                               dSet))
              (dimensioned 3 'ms-1))))
```

```
(event-trigger
  (duration 0 0 0 5)
  (lambda (dSet)
    (dim<? (average (filter (lambda (d) (body-speed? d))
                            dSet))
           (dimensioned 2 'ms-1)))))
```

In this example, it is assumed that `sprint-event` and `body-speed` are both specified as appropriate using the MADE archetype language; `sprint-event` is an observed event while `body-speed` is a measurement (with units ms^{-1}). Furthermore, the procedure `average` is assumed to be provided, although in practice, it must also be defined by the user in Rosette. The `average` procedure is applied to the `body-speed` measurements in the input data set and compared against the target value of $3\,\mathrm{ms}^{-1}$ for the start of the event and $2\,\mathrm{ms}^{-1}$ for the end of the event. Since the input data set may contain data of other types, the `filter` procedure, which is provided by Rosette, is used to ensure that only the body speed is considered.

7.2.3 Analysis Processes

As with that for Monitoring processes, the syntactic form for Analysis processes contains an identifier for the process, a Boolean proxy flag and an identifier for the output type. However, the syntactic form for Analysis processes also contains an arbitrary number of abstraction triplets, which in turn comprises a time window, an abstraction predicate and an abstraction function as specified in Sec. 6.3.3. Thus more formally:

```
analysis-process :
  '(define-analysis', ID, proxy, output-type,
      abstraction-triplet, {abstraction-triplet}, ')';
abstraction-triplet : '(', window, 2 * LAMBDA-EXPR, ')';
```

For example, the following syntax defines an Analysis process for determining whether the amount of exercise performed by a patient is insufficient or excessive. Insufficient is defined as less than 5 sprints per month (30 days) and excessive over 10 sprints per 7 days.

```
(define-analysis analyse-exercise #f exercise-abstraction
  ((duration 30 0 0 0)
   (lambda (dSet) (< (count-sprints dSet) 5))
   (lambda (dSet)
     (exercise-abstraction-value-space 'insufficient)))
  ((duration 7 0 0 0)
   (lambda (dSet) (> (count-sprints dSet) 10))
   (lambda (dSet)
     (exercise-abstraction-value-space 'excessive))))
```

It is assumed here that `exercise-abstraction` is specified as an abstraction archetype with the nominal values `'insufficient` and `'excessive`, the data type for which (i.e. `exercise-abstraction-value-space`) is automatically provided by the MADE archetype language. Furthermore, as with the `average` procedure described in Sec. 7.2.2, the procedure `count-sprints` must be defined by the user. In this case, `count-sprints` can be defined as follows:

```
(define-syntax-rule (count-sprints dSet)
  (length (filter (lambda (d)
                    (and (sprint-event? d)
                         (get-value (observed-event-value d))))
                  dSet)))
```

Apart from `get-value` and `observed-event-value`, which are provided by the MADE archetype language and is used to extract the value of an observed event, all other procedures (e.g. `define-syntax-rule` and `length`) are provided by Rosette. In effect, the procedure `define-syntax-rule` creates a procedure `count-sprints` that counts the number of data items in `dSet` that are sprint events with value true.

7.2.4 Decision Processes

Apart from the process identifier, boolean proxy flag, and output type identifier, the syntactic form for Decision processes also requires a non-empty list of instruc-

tion templates as well as a non-empty list of decision criteria (as dictated by its specification presented in Sec. 6.3.4):

```
decision-process :
  '(define-decision', ID, proxy, output-type,
    '(#:instructions', inst-template, {inst-template}, ')',
    '(#:criteria', LAMBDA-EXPR, {LAMBDA-EXPR}, ')', ')';
```

As implied by the syntactic form above, each decision criterion is a predicate expressed as a lambda expression. For instruction templates, three different syntactic forms are distinguished for specifying scheduled control instructions, scheduled homogeneous action instructions and scheduled culminating action instructions:

```
inst-template :
  control-template | homogeneous-template | culminating-template
```

The syntactic form for control templates comprises a symbol specifying its target process, a relative schedule for the process as well as a target status. Note that in accordance to the specification of Decision processes in the guideline model, the target schedule and status are both optional, but at least one must be present. Furthermore, the target status is represented as a boolean (with true denoting running and false paused), while relative schedules comprise multiple durations representing:

- The rounding factor with which to round the input date-time.
- The start pattern for the schedule.
- The repeat interval of the schedule, which can be a duration or a boolean (with true representing always repeating and false never).

```
control-template :
  '(control-template', SYMBOL, relative-schedule, status, ')' |
  '(control-template', SYMBOL, relative-schedule, ')' |
  '(control-template', SYMBOL, status, ')';
status : BOOLEAN;
relative-schedule :
  '(relative-schedule',
```

```
    '#:rounding', duration-instance,
    '#:pattern', duration-instance, {duration-instance},
    '#:interval', duration-instance | BOOLEAN, ')';
```

The syntactic form for homogeneous action templates comprises a symbol specifying its target action type, a relative schedule, an action rate (which is a dimensioned value) as well as an action duration:

```
homogeneous-template :
  '(homogeneous-action-template', SYMBOL, relative-schedule,
      dimensioned-instance, duration-instance, ')';
```

Finally, the syntactic form for culminating action templates comprises a symbol specifying its target action type, a relative schedule as well as a goal state, which can be an instance of any basic primitive value (the syntax for which is detailed in Sec. 5.2.1):

```
culminating-template :
  '(culminating-action-template', SYMBOL, relative-schedule,
      goal-state, ')';
goal-state :
  '(', ID, BOOLEAN | INTEGER | RATIONAL | SYMBOL, [SYMBOL], ')';
```

As an example, the following syntax specifies a Decision process to prescribe exercise if the patient is not performing sufficient amounts of exercise. The prescribed exercise repeats weekly and starts the day after the decision is made, and it alternates between running at $3\,\mathrm{ms}^{-1}$ for 30 s starting at 7:00 in the morning and running for 5000 m starting at 17:00 in the afternoon:

```
(define-decision
  decide-exercise #f exercise-plan
  (#:instructions
   (homogeneous-action-template
     'sprint-action
     (relative-schedule #:rounding (duration 1 0 0 0)
                        #:pattern (duration 0 7 0 0)
```

```
                        #:interval (duration 14 0 0 0))
    (dimensioned 3 'ms-1)
    (duration 0 0 0 30))
  (culminating-action-template
    'endurance-running-action
    (relative-schedule #:rounding (duration 1 0 0 0)
                       #:pattern (duration 7 17 0 0)
                       #:interval (duration 14 0 0 0))
    (dimensioned 5000 'm)))
 (#:criteria
   (lambda (dSet)
     (findf
       (lambda (d)
         (and (exercise-abstraction? d)
              (eq? (abstraction-value d)
                   (exercise-abstraction-value-space
                     'insufficient))))
       dSet))))
```

The procedure `findf` in the decision criteria is provided by Rosette, and it searches for a data item in the input data set that satisfies the given lambda expression. In this case, the lambda expression looks for a data item that is an exercise abstraction with the value representing insufficient.

7.2.5 Effectuation Processes

Similar to those for Monitoring, Analysis and Decision processes, the syntactic form for Effectuation processes contains an identifier for the process, a Boolean indicating whether it is a proxy process or not, and an identifier specifying the output type of the process. However, in accordance to its specification in the guideline model (Sec. 6.3.5), the syntactic form for Effectuation processes also contains a non-empty list of target schedules, each of which comprises:

- An identifier indicating the target plan type.

- A symbol indicating the target instruction type. For control instructions, this corresponds to the target process, while for homogeneous and culminating action instructions, this corresponds to the action type.
- A predicate on the target scheduled instruction, represented by a lambda expression.

```
effectuation-process :
  '(define-effectuation', ID, proxy, output-type,
      target-schedule, {target-schedule}, ')';
target-schedule :
  '(target-schedule', '#:plan', ID, '#:instruction', SYMBOL,
      '#:predicate', LAMBDA-EXPR, ')';
```

For example, the following Effectuation process is responsible for effectuating the action to run for 5000 m by activating the treadmill. In this case, the instruction predicate is specified to always return true since no extra conditions are to be attached to the effectuated action:

```
(define-effectuation effectuate-running #f treadmill-output
  (target-schedule #:plan exercise-plan
                   #:instruction 'endurance-running-action
                   #:predicate (lambda (inst-set) #t)))
```

7.2.6 MADE Process Instances

The syntactic forms presented above allow the various MADE processes in a guideline to be specified; to execute a guideline for a specific use case, these processes must first be instantiated. Like other data structures in Rosette, processes are instantiated as follows, with `ID` being the ID of the process to instantiate, the first `EXPR` evaluating to the data state of the process (i.e. a list of MADE data items) and the second `EXPR` evaluating the control state of the process:

```
  process-instance : '(', ID, EXPR, EXPR, ')';
```

For example, the following syntax creates an instance of `analyse-exercise` that contains an empty data state and is activated every 7 days starting from Mar.

3, 2020 at 7:00 am. Note that control state is implemented as a structure with the identifier `control-state` and two fields, the first for the process schedule and the second for the process status. Furthermore, the values of the data state and control state can be extracted using the procedures `made-process-data-state` and `made-process-control-state` respectively:

```
> (define proc-instance
     (analyse-exercise
      null
      (control-state
       (schedule (list (datetime 2020 3 3 7 0 0))
                 (duration 7 0 0 0))
       #t)))
> (made-process-data-state proc-instance)
'()
> (made-process-control-state proc-instance)
(control-state (schedule (list
  (datetime 2020 3 3 7 0 0)) (duration 7 0 0 0)) #t)
```

As with archetype instances, process instances can be compared against each other for equality using `eq?`, which checks whether two processes have the same ID or not and whether their data states and control states evaluate to the same values. Furthermore, `get-type` retrieves the Id of the process while `valid?` checks whether all components of a process, including its states and specification, are of the correct type or not:

```
> (eq? proc-instance (analyse-exercise null #f))
#f
> (eq? (get-type proc-instance) analyse-exercise)
#t
> (valid? proc-instance)
#t
> (valid? (analyse-exercise null #f))
#f
```

Apart from these procedures that are common between both MADE archetype and MADE process instances, the following procedures are also provided by the MADE guideline language for manipulating processes:

- `proxy?` which accepts a process as input and checks whether it is a proxy process or not.
- `is-proc-activated?` which checks whether a process is activated or not given its control state and date-time.
- `execute` which executes the given process with the given list of MADE data items at the given date-time.
- `generate-data` which returns the data generated by the input process with the given list of data at the given date-time.
- `update-data-state` which updates the data state of the input process with the given list of data and date-time.
- `update-control-state` which updates the control state of the input process with the given list of data and date-time.

The semantics of these procedures are all as described in Ch. 3. However, for simplicity, `update-data-state`, `update-control-state` and `generate-data` accept as input a complete process instance instead of its components as specified in Eq. 3.3 to Eq. 3.5. Furthermore, instead of outputting an updated data state and control state respectively, both `update=data-state` and `update-control-state` return as output a complete process instance, with the data state and control state updated as appropriate.

Example usage of these procedures are as follows; here, the input data (`in-data`) consists of two sprint events, one on Mar. 4, 2020 and the other on Mar. 13, 2020, and the date-time of execution (`cur-dt`) is Mar. 17, 2020 at 7:00 am. As expected, `generate-data` returns an insufficient exercise abstraction (lines 19 to 26), while `execute` returns an updated process together with the output data (lines 27 to 34). The new data state of the process is a list containing the input and output data (lines 35 to 40) and the control state of the process remains unchanged (lines 41 to 46).

```
1  > (define in-data
2      (list (sprint-event
```

```
            #f
            (datetime-range (datetime 2020 3 4 11 0 0)
                            (datetime 2020 3 4 11 0 30))
            #t)
           (sprint-event
            #f
            (datetime-range (datetime 2020 3 13 11 0 0)
                            (datetime 2020 3 13 11 0 30))
            #t)))
> (define cur-dt (datetime 2020 3 17 7 0 0))
> (proxy? proc-instance)
#f
> (is-proc-activated?
   (made-process-control-state proc-instance)
   cur-dt)
#t
> (define out-data (generate-data proc-instance in-data cur-dt))
> out-data
(list
 (exercise-abstraction
  #f
  (datetime-range (datetime 2020 3 17 7 0 0)
                  (datetime 2020 4 12 11 0 30))
  (exercise-abstraction-value-space 'insufficient)))
> (eq? (execute proc-instance in-data cur-dt)
       (list (update-control-state
              (update-data-state
               proc-instance (append in-data out-data) cur-dt)
              (append in-data out-data)
              cur-dt)
             out-data))
#t
> (eq? (made-process-data-state
```

```
        (update-data-state proc-instance
                           (append in-data out-data)
                           cur-dt))
       (append in-data out-data))
#t
> (eq? (made-process-control-state
        (update-control-state proc-instance
                              (append in-data out-data)
                              cur-dt))
       (made-process-control-state proc-instance))
#t
```

7.2.7 MADE Network Instances

Although a MADE network is specified to comprise a set of MADE processes (Eq. 3.7), the MADE guideline language provides for convenience the structure `made-network` to instantiate and manipulate MADE networks. This structure contains four fields, namely `monitoring` for a list of Monitoring processes, `analysis` for Analysis processes, `Decision` for Decision and `effectuation` Effectuation, and it supports the procedure `execute-network` for executing the given MADE network with the given input data at the given date-time. For example, the following syntax instantiates MADE network with the instance of `analyse-exercise` and executes the network with the sprint events as described in the previous section, with equivalent results. Note that `list-ref` is a procedure provided by Rosette for retrieving the contents of the given list at the given index; here it is used to, for example, extract the updated MADE network (line 7) and output data (line 11) resulting from `execute-network`.

```
1 > (define net-instance (made-network null
2                                      (list proc-instance)
3                                      null
4                                      null))
5 > (define net-exec-out
6     (execute-network net-instance in-data cur-dt))
```

```
> (eq? (made-network-analysis (list-ref net-exec-out 0))
       (list (list-ref (execute proc-instance
                               in-data cur-dt) 0)))
#t
> (eq? (list-ref net-exec-out 1)
       (list-ref (execute proc-instance in-data cur-dt) 1))
#t
```

7.2.8 Data List Generator

Apart from the syntactic forms presented above, the MADE guideline language also provides two solver-aided syntactic forms to support the verification of clinical guidelines: a data list generator presented in this section and a verifier in the next. Since MADE processes are modelled to operate on data sets, it is useful to instantiate multiple symbolic instances of MADE data items for verification and validation purposes. Therefore, a generator procedure is provided that can generate a list of instances of the given MADE archetype over the desired date-time period. More specifically, it accepts as input:

- An identifier specifying which archetype to instantiate.
- A date-time at which the MADE data items start to be generated.
- A date-time beyond which no MADE data items are generated.
- A repeat frequency, which can be either an integer to indicate the number of instances of the archetype to generate or a duration to indicate how often the archetype should be instantiated from the start date-time onwards.
- An optional list of target instruction types to include if an action plan is instantiated.

```
symbolic-archetype-instance-generator :
  '(generate-list', ID, 2 * datetime-instance, NATURAL |
      duration-instance, [target-list], ')';
target-list : '(list', SYMBOL, {SYMBOL}, ')';
```

As an example, executing the following syntax in Rosette will generate symbolic sprint events every two hours on March 17, 2020 from 00:00 am to 11:00 pm.

```
> (generate-list sprint-event
                 (datetime 2020 3 17 0 0 0)
                 (datetime 2020 3 17 23 0 0)
                 (duration 0 2 0 0))
```

7.2.9 Process Verifier

Analogous to the `verify-archetype` procedure for verifying archetypes, the procedure `verify-process` checks whether the input process is logically consistent or not. It requires as input the identifier for the process to be verified, an expression that evaluates to a list of list of archetype instances (which can be symbolic) as well as a symbolic or concrete date-time stamp at which to simulate the execution of the process:

```
process-verifier : '(verify-process', ID, EXPR, EXPR, ')';
```

The input list of archetype instances may be generated by combining the results of multiple `generate-list` procedures, while a symbolic date-time can be generated by executing the following `get-datetime` procedure, which is an extra function provided by the MADE guideline language to generate a date-time stamp between the specified start and end date-times. It accepts as input an optional pair of date-time stamps which indicate the range of permissible values for the year, month, day, hour, minute and second of the generated date-time stamp. If the date-time pair is not provided, then by default all generated date-time stamps are fixed to Dec. 15, 2019 `hr`:00:00, with the value of `hr` ranging from 0 to 23 inclusive.

```
symbolic-datetime : '(get-datetime, [2 * datetime], ')';
```

`verify-process` returns different results depending on the type of process being checked. For Monitoring processes of observed properties, it checks that executing the process at the given date-time stamp with the given data set would produce a valid observation. In other words, `verify-process` checks that the Monitoring process is indeed satisfiable. If yes, then it returns an example data set and date-time

stamp that results in a valid observation. Otherwise, it would return `(unsat)`, which indicates that either the specification of the Monitoring process is erroneous, or the input data set or date-time stamp is too restrictive. For Monitoring processes of observed events, `verify-process` checks the individual satisfiability of the trigger pairs for the start and end of an event. The conjunction of both trigger pairs are also checked; if satisfiable, this means that for some inputs, the Monitoring process may be non-deterministic, with its output dependent on whether the event start is detected first or the end.

For Analysis processes, `verify-process` also checks that the process can output a valid abstraction on execution (given the input data set and date-time). However, this satisfiability property is checked for each individual abstraction triplet in the specification. Furthermore, satisfiability is also checked for every pair of abstraction triplet to ensure that they are all mutually exclusive. If an example data set and date-time stamp can be found that satisfies more than one abstraction triplet, this means that for those inputs, the output of the Analysis process may be non-deterministic. That is, depending on the implementation of Analysis processes and/or the ordering of the abstraction triplets, a different abstraction may be generated given the same inputs.

Similarly, for Decision processes, `verify-process` checks that each individual decision criterion is satisfiable and that the process can output a valid action plan. However, unlike Analysis processes, combinations of decision criteria are not checked for satisfiability since the output of a Decision process remains the same regardless of which decision criterion is satisfied.

Finally, for Effectuation processes, `verify-process` checks the satisfiability of each target schedule as well as every possible pair of target schedules. If an input data set and date-time stamp satisfy two different target schedules, this means that the Effectuation process should output two instructions, one for each triggered schedule. However, this violates the specification of Effectuation processes, which can only output at most one instruction at each time instant. Thus this overlap may lead to indeterminate results depending on the implementation of Effectuation processes and/or ordering of the target schedules.

As an example, the following code checks the satisfiability of an Analysis process for exercise abstractions, which is specified in Sec. 7.2.3. The generator is used

to generate a list of 12 sprint events in 2-hour intervals on March 17, 2020 from 00:00 am to 11:00 pm, and the execution date-time for the process is set to be March 18, 2020 at 00:00 am.

```
> (verify-process analyse-exercise
                  (list (generate-list
                          sprint-event
                          (datetime 2020 3 17 0 0 0)
                          (datetime 2020 3 17 23 0 0)
                          (duration 0 2 0 0)))
                  (datetime 2020 3 18 0 0 0))
```

Since the Analysis process is specified to contain two different abstraction triplets, `verify-process` returns three results on execution, namely the inputs that satisfy the first abstraction triplet, the inputs that satisfy the second abstraction triplet, and the inputs that satisfy both. In this case, the abstraction triplets are mutually exclusive, thus the third result is `(unsat)`. For simplicity, only the main parts of the output are shown; the rest are replaced by `...` as a placeholder:

```
Model for abstraction triplet: 0
Input data:
(#(struct:sprint-event #f ...) ...)
Current date-time:
#(struct:datetime 2020 3 18 0 0 0)
Output data:
#(struct:exercise-abstraction #f
  #(struct:datetime-range #(struct:datetime 2020 3 18 0 0 0)
                          #(struct:datetime 2020 4 16 22 0 0))
  #(struct:exercise-abstraction-value-space insufficient))

Model for abstraction triplet: 1
Input data:
(#(struct:sprint-event #f ...) ...)
Current date-time:
```

```
#(struct:datetime 2020 3 18 0 0 0)
Output data:
#(struct:exercise-abstraction #f
  #(struct:datetime-range #(struct:datetime 2020 3 18 0 0 0)
                          #(struct:datetime 2020 3 24 2 0 0))
  #(struct:exercise-abstraction-value-space excessive))

Model for abstraction triplets: 0 and 1
(unsat)
```

7.3 Reference Implementation

7.3.1 Implementation of the Data Types

Similar to the MADE archetype language, implementing the MADE guideline language in Rosette involves implementing the constructs necessary for the MADE guideline model as well as the appropriate macros to transform the language syntax into the appropriate Rosette constructs. The source code of the implementation of the MADE guideline model can be found at https://github.com/nlsfung/made-language/tree/master/rpm; to summarise, it builds on top of the implementation for the MADE archetype language, with each data type specified in the MADE guideline model implemented as a structure in Rosette.

For example, recall that a generic process (Eq. 3.1) is modelled as follows:

$$Process = Id \times DataState \times ControlState \times InstSpec$$

This translates to the following in Rosette:

```
(struct made-process (data-state control-state))
```

The `struct` keyword declares a new structure type with `made-process` as the identifier, and it contains two fields: `data-state` and `control-state`. Note that since an identifier is automatically assigned to every new structure type, it is not necessary for the implemented MADE processes (and MADE data items) to contain a separate ID field to specify its type.

Furthermore, since `InstSpec` is modelled to remain constant over time, it is not implemented as a field but as procedures that return the appropriate values. Indeed, the proxy flag of a process is also implemented as a procedure that returns the appropriate boolean value (and can be called by the generic `proxy?` procedure). For example, recall that for Analysis processes (Eq. 6.14):

$$ASpec = OutputType \times \mathscr{P}(TimeWindow \times AbstractionPredicate \times AbstractionFunction)$$

This translates to the following generic interface for Analysis processes, which contain three methods:

- `analysis-process-output-type` to return the output type.
- `analysis-process-output-specification` to return the set of abstraction triplets.
- `analysis-process-proxy-flag` to return the boolean proxy flag.

```
(define-generics analysis
  [analysis-process-output-type analysis]
  [analysis-process-output-specification analysis]
  [analysis-process-proxy-flag analysis])
```

An implementation of `analyse-exercise` may then be as follows:

```
(struct analyse-exercise analysis-process ()
  #:methods gen:analysis
  [(define (analysis-process-output-type self)
      exercise-abstraction)
   (define (analysis-process-output-specification self) x)
   (define (analysis-process-proxy-flag) #f)])
```

The first line states that `analyse-exercise` is a structure that inherits from `analysis-process` (which, although not shown, inherits the `data-state` and `control-state` fields from `made-process`). The second line states the process

that `analyse-exercise` implements the `gen:analysis` interface for Analysis processes, and this is followed by the implementations of the three methods of the interface. In this example, the output type of the process is `exercise-abstraction` while `x` is a placeholder for the appropriate abstraction triplets. Since the process is not a proxy, the boolean proxy flag is false.

7.3.2 Implementation of the Invariants

All MADE process types are implemented as structures in Rosette as exemplified above; components are implemented as either fields in the structure or as interfaces. On the other hand, the function signatures and invariants that govern the behaviour of the MADE processes are converted into the corresponding procedures and procedure bodies in Rosette. For example, consider Inv. 3.2 which is reproduced below and states that if the process is not activated at the current date-time stamp (as determined by its control state), the process will not output any data items:

Invariant. Let s_{data} be an arbitrary data state, s_{ctrl} a control state, s_{inst} an instruction specification, d_{in} an input data set and t a date-time stamp. Then:

$$\neg isProcessActivated(s_{ctrl}, t) \Rightarrow generateData(s_{data}, s_{ctrl}, s_{inst}, d_{in}, t) = \{\}$$

Since this invariant relates to *generateData*, it translates to the following procedure body within its implementation (i.e. `generate-data`):

```
(if (is-proc-activated?
      (made-process-control-state made-proc)
      datetime)
    x
    null)
```

The first line marks the start of an `if-then-else` branch with the condition being the results of *isProcessActivated* (which is implemented as the procedure `is-proc-activated?` in Rosette). If the condition returns true, then *generateData* continues executing where placeholder `x` is. Otherwise, in accordance to the invariant, the function returns an empty list (`null`). Note that since Rosette does not support sets, they are implemented as lists instead.

Converting an arbitrary specification into an implementation is an undecidable problem, and it is not the intentions of this thesis to demonstrate a systematic and automatic method for translating invariants into source code. In fact, for some low-level operations (such as those for date-time comparisons), the specifications and implementations are assumed and not detailed in this thesis except in the source code. However, in general, invariants containing implications were implemented as `if-then-else` expressions in Rosette. Furthermore, quantifiers may be implemented using `foldl`, which folds a list starting from the left-most element, while sets can be constructed using a combination of `map`, `append`, `filter` and other similar operations.

7.3.3 Implementation of the Syntactic Forms

The implemented data structures and procedures as presented above provide the groundwork for the implemented macros which transform the syntactic forms of the guideline language into the appropriate Rosette constructs. The source code for the implemented macros are available at https://github.com/nlsfung/made-language/tree/master/lang; to summarise, each process specification is transformed into a structure that inherits from the appropriate MADE process data structure while re-implementing its interface. Clinical guidelines can then be formalized into a Rosette program by extending the structures and implementing the interfaces to return the appropriate values. Subsequently, during execution, the structures are instantiated as concrete data items.

As with the MADE archetypes, each specified process also re-implements the `typed` interface, such that `get-type` returns the identifier of the specified process and `valid?` checks whether instances of that process satisfy the specifications of the MADE guideline model. For example, recall the specification of the process for analysing exercise, which is reproduced below with `LAMBDA` acting as a placeholder for actual lambda expressions:

```
(define-analysis analyse-exercise #f exercise-abstraction
  ((duration 30 0 0 0) LAMBDA LAMBDA)
  ((duration 7 0 0 0) LAMBDA LAMBDA))
```

This is effectively expanded into the following source code in Rosette:

```
(struct analyse-exercise analysis-process ()
  #:transparent
  #:methods gen:analysis
  [(define (analysis-process-output-type self)
      exercise-abstraction)
   (define (analysis-process-output-specification self)
     (list (abstraction-triplet (duration 30 0 0 0)
                                LAMBDA LAMBDA)
           (abstraction-triplet (duration 7 0 0 0)
                                LAMBDA LAMBDA)))
   (define (analysis-process-proxy-flag self) #f)]

  #:methods gen:typed
  [(define/generic super-valid? valid?)
   (define (get-type self) analyse-exercise)
   (define (valid? self)
     (and (valid-spec? self)
          (super-valid?
            (made-process (made-process-data-state self)
            (made-process-control-state self)))))])
```

As expected, `analyse-exercise` inherits from `analysis-process` and re-implements its corresponding `gen:analysis` interface. The output type of the process is `exercise-abstraction`, and it contains two abstraction triplets, one with a time window of 30 days and the other 7. Furthermore, to support type-checking, `analyse-exercise` re-implements the `gen:typed` interface, with `get-type` returning the identifier `analyse-exercise` and `valid?` checking that the Analysis process contains a valid specification. The `super-valid?` procedure checks that the data state and control state of all instances of the process are valid, while `valid-spec?` checks whether the re-implemented methods return the appropriate results. Although not shown in the example above, for Analysis processes, this means ensuring that `analysis-process-output-type` returns an output type and that

`analysis-process-output-specification` indeed returns a list of abstraction triplets.

The syntactic forms for Monitoring, Decision and Effectuation processes are implemented similarly, the details of which can be found in the source code. For the data list generator, it is implemented as a procedure that recursively calls the appropriate archetype getter to obtain a list of MADE data items. More specifically, the implemented algorithm is as follows:

1. Obtain getter for the input MADE archetype.
2. If the repeat frequency is a duration:
   (a) Set current date-time to be the start date-time
   (b) If current date-time is past the end date-time, return an empty list.
   (c) Otherwise, call the archetype getter with the current date-time as input. This would generate an instance of archetype at the current date-time.
   (d) Set the start date-time to equal the current date-time plus the repeat frequency.
   (e) Call the data list generator again with the new start date-time.
3. If the repeat frequency is an integer:
   (a) If repeat frequency is 0, return an empty list.
   (b) Otherwise, call the archetype getter with the start and end date-times as input. This would generate an archetype instance between those date-times.
   (c) Decrement the repeat frequency by 1 and call the data list generator again.

To ensure that the output data list is meaningful, `generate-data-list` also asserts that all data items in the list are valid and unique, which is automatically enforced when executing Rosette's solver-aided facilities, including the `verify-process` procedure. In effect, given a list of possibly symbolic data items (`d-list`) and a possibly symbolic date time stamp (`dt`), invoking `verify-process` on a process (`p`) entails checking if executing `generate-data`, which is the implemen-

tation of *generateData*, can result in a valid MADE data item. In other words, `verify-process` is largely equivalent to the following code in Rosette:

```
(solve
  (assert
    (and (not (null? (generate-data p d-list dt)))
         (valid? (list-ref (generate-data p d-list dt) 0)))))
```

The main discrepancy is that `verify-process` checks individual portions of their specification instead of the complete process. For example, recall that executing `verify-process` on Analysis process checks the satisfiability of individual abstraction triplets as well as pairs of abstraction triplets. Thus in essence, `verify-process` creates simplified copies of a process `p`, each of which is checked independently using Rosette.

7.4 Verification of the Reference Implementation

Apart from guiding the implementation, the invariants were also translated into assertions for verifying the implementation using Rosette. This translation involves two steps:

- Defining the necessary symbolic constants to represent arbitrary real, integer or Boolean values for the assertion. Bitvectors and uninterpreted functions are also supported by Rosette, but they are not useful for verifying the reference implementation. Furthermore, arbitrary structures and lists, while useful, are not supported by Rosette, thus these must be instantiated in advance and populated with values (possibly symbolic) for Rosette to verify.
- Strengthening the invariant ensure that the resulting assertion for Rosette to verify is tractable.

For example, consider the implementation of Inv. 3.2 presented in Sec. 7.3.2. In theory, to verify that the body of `generate-data` satisfies Inv. 3.2, it is necessary to consider all valid control states, date-time stamps as well as substitutes for `x`. Now

recall that:

$$ControlState = Schedule \times Status$$
$$Schedule = \mathscr{P}(DateTime) \times RepeatInterval$$
$$RepeatInterval = Duration \cup Boolean$$

Since arbitrary lists are not supported, the assertion can be simplified to only verify schedules with exactly one date-time stamp. In fact, a date-time stamp contains six different integers representing the year, month, day, hour, minute and second, so to ensure that the verification process is tractable, it may be decided that for all date-time stamps, only one field (e.g. the hour) is assigned a symbolic value while the others are assigned concrete values (e.g. 2019-12-12 `hr`:30:15). To simplify the assertion even further, it may also be decided to set the control state status to true (so that the process is running) and to assign the repeat interval a symbolic Boolean value instead of a duration (which is another structure comprising 4 integers). Finally, the data state of the process and the data set input into `generate-data` may both be assumed to be empty, but to ensure that *generateData* can produce a non-empty data set, the substitute for `x` may be an arbitrary data item. This results in the following strengthened version of Inv. 3.2:

Invariant. Let t_1 and t_2 be arbitrary natural numbers (ranging from 0 to 23 inclusive), b an arbitrary boolean value and s_{inst} an instruction to return TRUE. Furthermore, let $s_{ctrl} = ((\{(2019, 12, 12, t_1, 30, 15)\}, b), \text{TRUE})$ and $t = (2019, 12, 12, t_2, 30, 15)$. Then:

$$\neg isProcessActivated(s_{ctrl}, t) \Rightarrow generateData(\{\}, s_{ctrl}, s_{inst}, \{\}, t) = \{\}$$

In Rosette, this can be implemented as follows, with `generate-data-x` being a special implementation of *generateData* that returns true when the process is executed:

```
(define-symbolic hr1 integer?)
(define-symbolic hr2 integer?)
(define-symbolic b boolean?)
```

```
(define c-state
  (control-state
    (list (datetime 2019 12 12 hr1 30 15)) b))
(define t (datetime 2019 12 12 hr2 30 15))
(verify
  (assert
    (implies (not (is-proc-executed? c-state t))
             (null? (generate-data-x null c-state null t)))))
```

On executing the `verify` method, Rosette returns `(unsat)` to inform the user that the input assertion is valid. Otherwise, Rosette returns a counter-example that violates the assertion. In this way, all 38 invariants are successfully verified, the assertions for which are available at https://github.com/nlsfung/made-language/tree/master/inv.

Note that since these invariants relate to the specification of MADE processes only, the implementations for the data list generator and process verifier were not formally verified. Instead, they were checked by testing on example processes, such as the Analysis process `analyse-exercise` introduced in Sec. 7.2.6, the results for which are presented in Sec. 7.2.8 (for `generate-list`) and Sec. 7.2.9 (for `verify-process`) and are manually verified to be correct. Such testing procedure was also performed on an example Monitoring, Decision and Effectuation process, the details of which are available at https://github.com/nlsfung/MADE-Language/blob/master/exp/TestProcesses.rkt and included in Appendix B.

7.5 Discussion

7.5.1 Comprehensibility of the Language

One typical feature of knowledge representation languages is that it is not only computer-interpretable, but also human-readable; this allows the user to easily inspect the stored knowledge and understand the reasoning performed by the knowledge-based system. Therefore, although it is not the purpose of this thesis to produce a commercial knowledge representation language and system, it is useful to discuss the comprehensibility of the MADE guideline language and possible improvements that can be made.

In particular, the MADE guideline language relies heavily on the syntax of Rosette (and Racket), especially for defining lambda expressions, thus it may not be accessible to users that are not already familiar with these programming languages. To address this issue, extra syntactic forms can be introduced to raise the level of abstraction of the MADE guideline language, such that lambda expressions can, for example, be expressed using the infix notation typical of mathematical and logical expressions (instead of the prefix notation adopted by Rosette). To improve re-usability, the syntax of existing languages for the medical domain may also be adopted. For example, in 1997, Shahar presented a knowledge-based method for performing temporal abstractions [67], the syntax for which may be adopted for specifying Analysis processes. Similarly, the GELLO expression language, which is standardised by HL7 [15], may also be adopted in place of arbitrary lambda expressions. However, this requires the semantics of the languages to be equivalent to avoid confusion.

7.5.2 Utility of the Solver-Aided Forms

It is outside the scope of this thesis to perform a full evaluation of the solver-aided syntactic forms , but it can be observed that the process verifier is sound, i.e. if an error is detected, then there is definitely an error in the specified process. For example, if a data set can be found that satisfies two different abstraction triplets in an Analysis process, then it is clear that under those circumstances, the output of that Analysis process will be non-deterministic since more than one abstraction triplet is triggered. However, this assumes that there is no redundancy in the specified processes. For example, it may be the case that the overlapping abstraction pairs are equivalent to each other, in which case the same outputs would be produced regardless.

Furthermore, it can be observed that the process verifier is incomplete. In other words, even if no errors are detected, it can not be guaranteed that the process is free from errors. In fact, one key limiting factor for the process verifier is that it requires the user to provide a list of symbolic data items to test. If the list is too restrictive, then no errors will be detected. On the other hand, if the list is too flexible (i.e. it includes too many symbolic constants), then Rosette may not terminate in a reasonable amount of time or it may run out of memory. Thus, the user must be

familiar with both the specified process and the limitations of Rosette to arrive at an appropriate data set for the verifier.

Chapter 8

Case Study

8.1 Introduction

In previous chapters, the underlying models and resulting languages for representing MADE data and processes were presented. A reference implementation of the languages was also developed, tested and formally verified using Rosette. In this chapter, the MADE languages are validated by applying them to a clinical guideline, specifically the one for gestational diabetes mellitus (GDM) [22] that was developed as part of the MobGuide project to validate the MobiGuide system (Sec. 1.4).

Defined as glucose intolerance that started or was first recognized during pregnancy [1], gestational diabetes occurs in approximately 7 % of pregnancies [1] and may lead to complications such as excessive birth weight, future diabetes and respiratory distress for the newborn child [46]. To prevent these adverse outcomes, patients are required to monitor their blood glucose levels regularly and frequently, the results of which are then traditionally reviewed during outpatient visits. This places a burden on the hospital as well as inconveniencing the patient [48], thus various studies have been conducted on the benefits of telemedicine on patients with gestational diabetes. An example of such research is by Garcia-Saez et al., who presented the GDM guideline [22] that was adopted by the MobiGuide project [57].

To test its clinical applicability, the MADE languages are used to formalize this GDM guideline into a program that can be executed and analysed using Rosette. In Sec. 8.2, an overview of the clinical guideline is presented, followed by an explanation

of the adopted procedure for validating the MADE languages in Sec. 8.3. The results are presented in Sec. 8.4 and discussed in Sec. 8.5.

8.2 Guideline Overview

The full gestational diabetes guideline on which the MADE languages were applied was developed as part of the MobiGuide project and released by Goldstein et al. in 2014 [25]. To summarise, it originated from a narrative guideline produced by Rigla et al. in 2013 for MobiGuide [61], from which the fragment shown in Fig. 3.4 was extracted. However, the full guideline extends the narrative version by including more details on how GDM patients can be supported during their daily lives [22]. The result is 13 different semi-formal workflows, such as that shown in Fig. 8.1 for managing urinary ketone levels. The dashed back border indicates the part of the workflow that can be derived directly from the guideline fragment shown in Fig. 3.4; everything outside the bordered area are extensions to that fragment.

In general, each workflow starts with a measurement action (blue rectangle) and ends with one or more plans (orange rectangles), each of which is a placeholder for another workflow or a recommendation for the patient or clinician. For Fig. 8.1, the workflow is executed after every measurement of ketonuria (specifically of urinary ketone levels), and it may lead to decisions such as monitoring ketonuria twice weekly (`MONITOR TWICE/WEEK`), doing nothing and increasing carbohydrates at dinner. Each of these decisions are predicated on specific conditions that are represented by the blue or orange diamonds; for example, the figure shows that if the measured urinary ketone levels is not positive and is in fact negative for two weeks, then this will lead to the decision to monitor ketonuria twice weekly (which is another workflow). Note that starting a new workflow generally implies terminating the current one, and some details, such as the definition of “positive” ketonuria, are inferred from other parts of the guideline or from the narrative version of the guideline.

All 13 semi-formal workflows in the guideline share the same characteristics as exemplified in Fig. 8.1, and their full details are reproduced in Appendix C. By formalising them using the MADE languages, these workflows serve as input to validate the clinical applicability of the MADE languages and the underlying models.

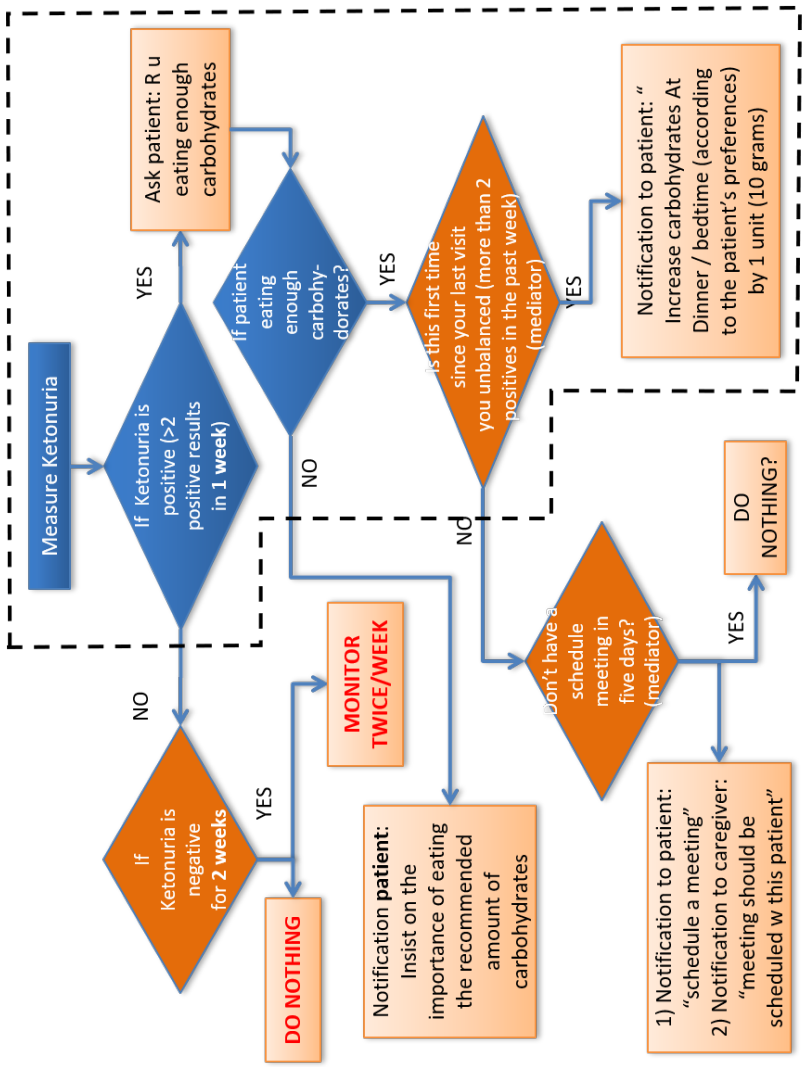

Fig. 8.1 A typical workflow reproduced from the GDM (gestational diabetes mellitus) guideline [22], with an additional dashed black border delineating the parts of the workflow that originate directly from the narrative guideline [61].

8.3 Validation Procedure

8.3.1 Formalization of the Guideline

As shown previously in Fig. 1.4, the adopted procedure for validating the MADE languages can be roughly divided into two steps. The GDM guideline is first formalised using the MADE languages into a Rosette program, which is subsequently verified using Rosette to not contain any unexpected errors. While it is beyond the scope of this thesis to develop a comprehensive methodology for formalising clinical guidelines, the process of formalising the GDM workflow was guided by the following seven rules, which were generally applied on the workflows in order.

Rule 1. Each measurement action becomes a Monitoring process. Thus, the action to measure ketonuria in Fig. 8.1 may be formalised into a process for monitoring urinary ketone levels.

Rule 2. The target of the measurement action becomes an observation archetype. This means that from the action to measure ketonuria, urinary ketone observations can be derived, which is an observed property with five grades as specified in the guideline extract (Fig. 3.4): - -, -, +/-, + and + +. Although the workflow implies that urinary ketone level is a measurement instead of observation, it is modelled as an observation instead because it has a direct clinical interpretation and is directly used in the workflow to determine which plans to execute.

Rule 3. Each condition in the workflow becomes an abstraction triplet (in an Analysis process). As an example, consider the condition of negative ketonuria for two weeks (which leads to the decision to monitor ketonuria twice weekly). This condition translates to an abstraction triplet with a time window of two weeks, an abstraction predicate that returns true if and only if all input urinary ketone levels are negative (i.e. - - or -), and an abstraction function that returns the value “negative” (for ketonuria abstractions).

Rule 4. Each identified abstraction triplet is associated with an Analysis process and abstraction archetype. Depending on the other conditions in the clinical guideline, the abstraction triplet for negative ketonuria may become its own Analysis process, or it may be grouped together with other abstraction triplets to form an Analysis

process. In any case, this abstraction triplet also implies the necessity of a "ketonuria" abstraction with at least one possible value, namely "negative"; other abstraction values may be implied by other abstraction triplets.

Rule 5. Each identified abstraction triplet is associated with a decision criterion in at least one Decision process. For the ketonuria abstraction, only one Decision process is required to detect whether ketonuria has been negative for two weeks; this process is responsible for deciding to monitor ketonuria twice weekly. If negative ketonuria can lead to other decisions, then it should be incorporated into other Decision processes as appropriate.

Rule 6. Each plan in the workflow becomes an action plan in the formalized guideline. Returning to the negative ketonuria example, the resulting action plan is to monitor ketonuria twice weekly, which although not shown in Fig. 8.1, is a different workflow. As a result, this action plan contains control instructions to start all the MADE processes derived from the other workflow. Furthermore, by executing another workflow, it is implied that the current workflow is no longer active, thus the action plan should also contain control instructions for disabling or rescheduling the corresponding MADE processes (depending on the requirements of the new workflow and other remaining workflows).

Rule 7. Each identified action plan is associated with the necessary Effectuation processes and instruction archetypes to effectuate it. For example, to disable the workflow shown in Fig. 8.1, the appropriate control instructions and Effectuation processes may be required to disable the Analysis process for generating ketonuria abstractions as well as the Decision process for monitoring ketonuria twice weekly. The Monitoring process for urinary ketone levels (ketonuria) may also require rescheduling depending on the requirements of the new workflow.

8.3.2 Verification of the Formalised Guideline

It is beyond the scope of this thesis to validate that the formalized guideline is a clinically accurate translation of the workflows. However, the formalized guideline was tested and verified using the `verify-archetype` and `verify-process` procedures to help assure that it is free from potential run-time errors.

As described in Sec. 5.2.9, `verify-archetype` is a procedure provided as part of the MADE archetype language to check whether an archetype is logically consistent. If any of the MADE archetypes in the formalised guideline contain inconsistencies, such as a measurement value that must be greater than 50 and less than 0, `verify-archetype` would return `(unsat)`; otherwise, a concrete instance of the archetype would be returned.

Similarly, as explained in Sec. 7.2.9, `verify-process` is a procedure provided as part of the MADE guideline language to check the satisfiability of MADE processes. Thus, for each MADE process derived from the GDM guideline, `verify-process` is used to check whether a set of input data can be found that would result in the output of a data item. Depending on the type of process being verified, the procedure also checks whether the MADE process is completely unambiguous. However, to verify a MADE process, `verify-process` also requires the input of an appropriate set of symbolic input data, which must be manually derived by inspecting the guideline. For example, since the decision to monitor ketonuria twice weekly is conditional on the detection of negative ketonuria, at least one symbolic ketonuria abstraction must be provided to verify this Decision process.

8.4 Validation Results

8.4.1 Formalised Guideline

The full details of the formalized guideline, which total around 2700 lines or 90 pages, can be found online at https://github.com/nlsfung/MADE-Language/tree/master/exp/gdm and are divided into two files: `GdmIM.rkt` which contains the archetypes that constitute the domain information model for GDM, and `GdmPM.rkt` which contains the MADE processes that constitute all the tasks specified in the GDM guideline. To summarise, the formalized guideline comprise a total of 55 processes, specifically 0 Monitoring, 4 Analysis, 22 Decision and 29 Effectuation processes, all of which are connected together by the flow of 50 types of MADE data, specifically 0 Measurement, 8 Observation, 8 Abstraction, 14 Action Plan, 3 Action Instruction and 17 Control Instruction archetypes. An overview of all the MADE archetypes and MADE processes derived from the GDM guideline can be

found in Appendix D; as an example, shown below are the archetypes for urinary ketone observations (`urinary-ketone`) and ketonuria abstractions (`ketonuria`), together with the specification of the process for analyzing urinary ketone levels (`analyse-urinary-ketone`):

```
(define-observation urinary-ketone enumerated
  '-- '- '-/+ '+ '++)

(define-abstraction ketonuria enumerated
  'negative 'positive)

(define-analysis analyse-urinary-ketone #f ketonuria
  ((duration 7 0 0 0)
   (lambda (d-list)
     (let* ([uk-list (filter (lambda (d) (urinary-ketone? d))
                              d-list)])
       (andmap (lambda (d)
                 (enum<? (observed-property-value d)
                         (urinary-ketone-value-space '-/+)))
               uk-list)))
   (lambda (d-list) (ketonuria-value-space 'negative)))
  ((duration 7 0 0 0)
   (lambda (d-list)
     (let* ([uk-list (filter (lambda (d) (urinary-ketone? d))
                             d-list)])
       (> (rosette-count
           (lambda (d)
             (enum>? (observed-property-value d)
                     (urinary-ketone-value-space '-/+)))
           uk-list)
          2)))
   (lambda (d-list) (ketonuria-value-space 'positive))))
```

As expected, the example above conforms to and extends the formal specifications presented in Sec. 4.4 and Sec. 6.4, specifically Eq. 4.33 for the specification of urinary ketone levels, Eq. 4.34 for ketonuria abstractions and Eq. 6.32 for the process for analysing ketonuria. For example, as mentioned in Sec. 4.4, the value space for urinary ketone levels (line 2) was inferred from the clinical guideline to contain the values `- -`, `-`, `+/-`, `+` and `++`. Furthermore, it was inferred from the workflow that ketonuria abstractions exhibit two values (line 5): negative and positive. This is leads to two abstraction triplets in the process for analysing ketonuria, one for each possible abstraction value:

- If urinary ketone levels are all negative for one week, then ketonuria is negative (lines 8 to 16).
- If there are more than two positive urinary ketone levels in one week, then ketonuria is positive (lines 17 to 27).

8.4.2 Verification Results

The `verify-archetype` procedure was used to verify each MADE archetype for the GDM guideline; as expected, each archetype was satisfiable and can be instantiated into a concrete data item. For example, the following was the result of verifying `urinary-ketone`:

```
Example #<procedure:urinary-ketone>:
#(struct:urinary-ketone #f #(struct:datetime 2019 12 15 0 0 0)
#(struct:urinary-ketone-value-space --))
```

Similarly, the `verify-process` procedure was used to verify each MADE process for the GDM guideline, the results of which were also as expected. The inputs used to verify each MADE process are detailed in Appendix E, and as an example, the following code was used to verify `analyse-urinary-ketone`:

```
> (verify-process
   analyse-urinary-ketone
   (list (generate-list
```

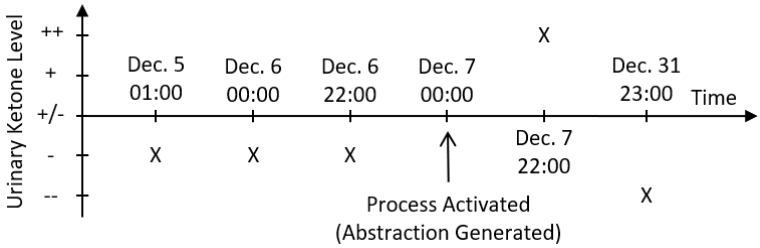

Fig. 8.2 The data set generated by `verify-process` to demonstrate the satisfiability of the first abstraction triplet of `analyse-urinary-ketone`, which is the detection of negative ketonuria levels in the past week.

```
        urinary-ketone
        (datetime 2019 12 1 0 0 0)
        (datetime 2019 12 31 24 0 0)
        5))
  (get-datetime (datetime 2019 12 1 0 0 0)
                (datetime 2019 12 31 24 0 0)))
```

The data generator (`generate-list`) was used to generate a list of 5 symbolic `urinary-ketone` instances, each with a valid date-time between Dec. 1 and Dec. 31, 2019. Furthermore, the execution date-time for the process was also set to lie within the same range, and these inputs were used by `verify-process` to check whether for `analyse-urinary-ketone` each single abstraction triplet and each pair of abstraction triplets are satisfiable. As expected and as shown in Fig. 8.2 and 8.3, an example data set was generated for each abstraction triplet, thus demonstrating their satisfiability. Furthermore, no data set was found that satisfies both the first and second abstraction triplet, which implies correctly that the two abstraction triplets are mutually exclusive. For cases in which the abstraction triplets are not mutually exclusive, the abstraction triplets may be strengthened or separated into different processes, the appropriate choice for which depends on the specific clinical requirements.

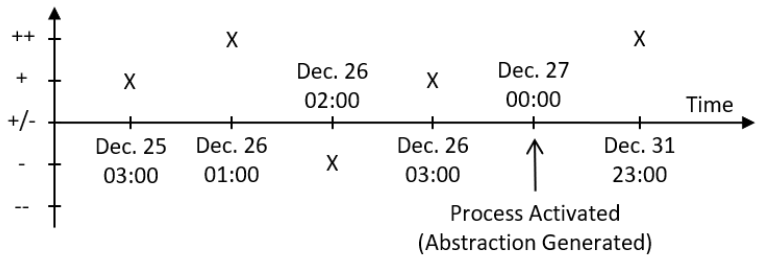

Fig. 8.3 The data set generated by `verify-process` to demonstrate the satisfiability of the second abstraction triplet of `analyse-urinary-ketone`, which is the detection of positive ketonuria levels in the past week.

8.4.3 Other Observations

All automatable portions of the clinical guideline was successfully formalised and verified using the MADE languages. However, it is useful to note that some non-automatable portions of the guideline were not formalized. An example is the workflow for deciding the appropriate hypertension plan for the patient (Fig. C.8), which must be executed by the clinician in a hospital setting and is therefore outside the scope of the MADE languages. Indeed, although measurement actions in the workflow clearly correspond to Monitoring processes, zero measurement archetypes and zero Monitoring processes were formalized as the original guideline does not specify what the raw measurements are and how they should be processed.

Furthermore, of the 51 action points (orange rectangles) in the guideline, 15 were not formalized explicitly as they represent simple notifications about the state of the patient instead of a recommendation to perform certain actions. Returning to the ketonuria workflow for example, the notification to schedule a meeting is not modelled explicitly since it is not clinically relevant per se but merely serve as an indication of an underlying, clinically relevant situation. In this case, the clinically relevant condition is that the patient exhibits positive ketonuria while being compliant with her prescribed carbohydrates intake level. On the other hand, while increasing carbohydrates dinner intake also involves a notification to the patient, it is modelled explicitly (as a culminating action) since the notification relates to a distinct and clinically significant concept.

Finally, apart from or in addition to starting a new workflow, certain action points in the workflow can modify future executions of the same workflow. For example, as indicated by the workflow shown in Fig. 8.1, the decision to increase carbohydrates intake should only be made the first time the patient exhibits positive ketonuria while being compliant with her diet prescription; subsequent occurrences of the same condition requires the patient to schedule a meeting with the clinician. Such workflows can be formalized by first decomposing them into multiple children workflows, with one child activating another. Thus returning to the example, the ketonuria workflow can be decomposed into two workflows, the first for increasing carbohydrates intake and the second for scheduling a meeting (which is activated by the first). In this way, the decomposed workflows can be formalized as usual by following the rules presented in Sec. 8.3.1.

8.5 Discussion

8.5.1 Clinical Appropriateness of the MADE Languages

Previously, it has been argued that the MADE languages are as expressive as conventional guideline representation languages; in this chapter, it is demonstrated using a clinical guideline for gestational diabetes mellitus that the MADE languages can indeed capture all the knowledge required to automate the provision of pervasive support to patients. However, it was noted that certain non-automatable aspects of the guideline cannot be formalized, most notably the measurement actions in the guideline workflows.

While these measurement actions are intended to be performed outside the system (by the patient using off-the-shelf sensors), it may nevertheless be useful to capture such knowledge to provide contextual information for the rest of the formalized guideline. For example, the definition of positive ketonuria (i.e. more than 2 positive urinary ketone levels) may in fact depend on the exact frequency of measurements performed by the patient (once every day). One simple solution to include this knowledge is to add comments to the formalized guideline (which is supported by Rosette), but a more formal approach may be more appropriate to support the verification and testing of formalized guidelines.

Furthermore, the MADE languages in their current form are unable to capture simple user notifications, such as those to remind the patient to perform a task. Indeed, it is also not possible to personalise the formalised guideline according to patient preferences, which is an important usability feature for providing knowledge-based decision support to patients [57]. For example, dinnertime must be made explicit (e.g. 7 pm) to formalize the decision process to increase carbohydrates intake; individual adjustments can only be made by directly accessing and changing the formalized guideline before deployment. For more complicated tasks, such as the administration of drugs, other "algorithmic" knowledge that cannot be expressed using the MADE languages may be necessary to adjust the properties of the task (e.g. dosage) depending on the patient's specific circumstances.

Although such knowledge is clearly useful, they are outside the scope of the MADE formalism since its purpose is to capture the relevant clinical processes in a clinical guideline and enable their distribution and parallel execution for pervasive support. However, the MADE formalism may be augmented with other existing languages that are designed specifically to capture such kinds of knowledge. For example, Klimov and Shahar in 2013 presented the iALARM language which is explicitly designed for managing alerts [41]. Similarly, Florence et al. in 2015 presented a patient-oriented prescription programming language (POP-PL), which is a language for specifying prescriptions, including instructions for modifying drug dosage [20].

8.5.2 Utility of the Solver-Aided Features

For this thesis, the `verify-archetype` and `verify-process` procedures were mainly used to assure that no programming errors were present in the formalized clinical guideline. While it is beyond the scope of this thesis to fully evaluate the usefulness of these procedures in validating the clinical correctness of the formalized guidelines, these procedures have successfully detected overlaps within the specifications of individual processes (such as between pairs of abstraction triplets). However, one clear limitation of these procedures is that they can only verify individual archetypes and individual MADE processes but not complete MADE networks. Manual testing is still required to verify complete guidelines, which may not be

the most appropriate given the potential complexity of clinical guidelines and the resulting MADE networks. For example, it may be useful to check that two mutually exclusive Decision processes will never be activated at the same time, but this can never be fully guaranteed with testing alone.

Therefore, it is useful to extend the MADE languages to include procedures for verifying MADE networks, but unlike verifying individual processes, verifying MADE networks may involve intractably large streams of data. In fact, attempts at verifying the invariants for the MADE network all proved to be intractable except for small networks and a few data items. As a result, verifying MADE networks may require raising the level of abstraction by removing irrelevant details about the specific data items and MADE processes to be verified. For example, the problem of verifying MADE networks may be reduced to a more abstract, state transition problem, which may then be solvable using state-of-the-art model checking algorithms, such as IC3 [9].

Chapter 9

Conclusions

9.1 Introduction

Guideline-based pervasive healthcare systems can extend evidence-based healthcare beyond the traditional healthcare setting. It is all the more crucial then to have demonstrable system resilience, quality of clinical information and correct operational logic in a highly distributed environment. To this end, the MADE guideline language (and the accompanying MADE archetype language) were developed to represent clinical guidelines in the context of pervasive healthcare. In Sec. 9.2, the research questions presented in Sec. 1.3 are reviewed in light of the research presented in this thesis. This is followed by a review and discussion of the general research contributions of this thesis in Sec. 9.3. Finally, possible future research directions and future outlook are presented in Sec. 9.4 and 9.5 respectively.

9.2 Research Questions

Main RQ. How can pervasive and knowledge-based support be provided to patients?

This thesis focuses on the provision of guideline-based support to patients, which traditionally assumes a fixed and centralized system architecture for executing clinical guidelines. On the other hand, components of a pervasive healthcare system may require dynamic reconfiguration in response to factors such as changing clinical

requirements, evolving patient preferences and unreliable communications environments. Therefore, in Ch. 3, a new architectural model is presented for representing disease management as a network of four types of data flow processes: Monitoring (M), Analysis (A), Decision (D) and Effectuation (E). These processes are modelled to execute in parallel such that they can be flexibly distributed across system components and provide support independently of each other, thereby avoiding a single point of failure.

This architectural model was formalized and developed into an archetype language (Ch. 4 and Ch. 5) as well as a guideline language (Ch. 6 and Ch. 7) to represent the data and processing requirements in clinical guidelines. Furthermore, the appropriateness of the MADE models and languages were all validated in Ch. 8 by formalizing a complete clinical guideline (for gestational diabetes mellitus). Since only data and not control is communicated between the processes, live locks and dead locks are guaranteed to not occur. In addition, arbitrary feedback loops are not permitted in the MADE model, thus the execution of a formalised guideline will always terminate provided that each individual MADE process terminates. However, as explained in Sec. 1.5, the MADE models and languages do not address the technical performance issues that might be affect pervasive healthcare systems. These include the potential need to process large data streams in real-time as well as the specific mechanisms with which a pervasive healthcare system can re-distribute its functionality.

RQ 1. What is an appropriate knowledge representation language for formalising clinical guidelines for pervasive healthcare systems?

The knowledge representation language presented in this thesis is derived in two stages from an architectural model presented in Ch. 3 for representing disease management. In the first stage, which is presented in Ch. 6, the MADE processes that constitute the architectural model were given detailed and formal semantics using axiomatic set theory. Thus, each type of MADE process was specified as a set with specific properties, and their behaviour were specified using function signatures and logical invariants. From this guideline model, which comprises a total of 28 set definitions, 13 function signatures and 38 logical invariants, the syntax and

semantics of the guideline representation language were derived in the second stage and presented in Ch. 7.

The appropriateness of the language was validated in Ch. 8 by formalizing the complete GDM guideline into a network of 55 MADE processes, all of which were verified to behave as expected using the solver-aided facilities of Rosette. Since MADE processes do not allow arbitrary branching, they are all guaranteed to terminate. Furthermore, evidence from the formalising the GDM guideline suggests that the MADE guideline language is sufficiently expressive to formalise all automatable parts of clinical guidelines. However, it was observed that non-automatable parts cannot be formalised, such as manual measurements of blood glucose. Customisation of clinical guidelines according to patients' personal preferences are also currently not supported by the MADE language.

RQ 2. What is an appropriate representation for patient data in the context of pervasive healthcare?

As with the representation for clinical guidelines, the representation for patient data is directly derived from the architectural model presented in Ch. 3, but by giving detailed and formal semantics to the data flow instead of the MADE processes. Six types of MADE data were distinguished, namely Measurement, Observation, Abstraction, Action Plan, Action Instruction and Control Instruction, each of which was formally specified using axiomatic set theory in Ch. 4. The resulting MADE reference information model comprises 32 set definitions, and it was in turn developed into an archetype language in Ch. 5 to enable the specification of sub-types of MADE data.

In Ch. 8, the appropriateness of the language was validated by formalizing the complete GDM guideline, deriving from it 50 MADE archetypes. It was observed that the MADE reference information model is comparable to similar models proposed by openEHR and HL7. However, it was also noted that since this research focuses on the clinical concerns of guideline-based pervasive healthcare, issues relating to quality-of-data are ignored, which are particularly pertinent due to the uncontrolled nature of the daily living environment.

RQ 3. What is an appropriate design and implementation of the guideline execution engine for guideline-based pervasive healthcare systems?

It is outside the scope of this thesis to produce a commercial guideline-based pervasive healthcare system; instead, this thesis aims to produce a reference implementation which can serve as a gold standard for future implementations that better reflect the non-functional requirements of pervasive healthcare systems. To achieve this, the implementation of the language was derived directly from the formal specifications of the MADE data and MADE processes presented in Ch. 4 and Ch. 6. The results, which comprise a set of libaries on top of Rosette, were presented in Ch. 5 and Ch. 7 respectively.

For the reference implementation, appropriateness is determined by its preservation of the semantics of the MADE models. Thus as presented in Ch. 7, Rosette's solver-aided facilities were used to assure that the implementation complies with the 38 logical invariants that govern the behaviour of MADE processes. Finally, Ch. 8 demonstrates how the implemented libraries can be used to formalise a complete guideline into a Rosette program that can be executed and formally verified.

RQ 4. What extensions can be incorporated into the guideline execution engine (and associated language) to support the verification and validation of formalised clinical guidelines?

Formalizing clinical guidelines is a non-trivial task. Therefore, while it is outside the scope of this thesis to develop a knowledge acquisition tool, the MADE languages were extended with constructs to support the formal verification of the formalized guidelines as demonstrated in Ch. 8. In Ch. 5, the MADE archetype language was presented which allows constraints on sub-types of MADE data to be specified. These constraints can be checked for consistency using the `verify-archetype` procedure and, if verified, can be used to automatically generate example data items using the appropriate getters. In turn, these generated data can be used to simulate the execution of a guideline and to formally verify individual MADE processes by using the `verify-process` procedure described in Ch. 7.

9.3 Research Contributions

The research presented in this thesis has contributed to both the theoretical and applied areas of science in the context of pervasive healthcare. More specifically:

- The MADE computational independent model (CIM) was developed and presented in Ch. 3 for representing the data flow in disease management. This model was used in the thesis to develop a guideline representation language for guideline-based pervasive healthcare systems, but because it does not assume any specific application or system infrastructure, it may also provide a useful framework for analysing and designing other types of pervasive healthcare systems. For example, the MADE CIM may be used as a general framework to elicit the functional requirements of pervasive healthcare systems and to understand their deployment across the available system components.
- To represent the data flow between MADE processes, a verified reference information model (RIM) and archetype language was developed and presented in Ch. 4 and Ch. 5 respectively. While the MADE RIM and archetype language forms part of the overall MADE guideline representation language, they can be applied independently of the guideline language to model clinical data for other types of pervasive healthcare systems. In this way, interoperability between the different systems can be achieved, such that different components of different systems may be easily integrated together to provide specific decision support functionality as required by each different clinical application (or combinations of applications).
- The MADE guideline language was developed and presented in Ch. 6 and Ch. 7 to formalize clinical guidelines in the context of pervasive healthcare. Unlike typical guideline representation languages which focus on the control flow between tasks, the MADE guideline language models the data flow between tasks, thereby removing the need for a centralized supervisory component to control the execution of formalized guidelines. Since models similar to the MADE CIM have been developed to represent intelligent agents and autonomous systems, the MADE guideline language may be applicable to other domains as well (such as for managing an industrial process instead of a patient).
- In Ch. 5 and Ch. 7, the reference implementation of the MADE archetype language and the MADE guideline representation language is presented. The reference implementation was verified against the specifications of the MADE

data and MADE processes using Rosette and is designed to serve as a gold standard from which more optimal implementations can be derived and compared against. Furthermore, by making use of Rosette's capabilities to execute and analyse programs, the reference implementation also provides a means for testing and verifying guidelines that are formalized using the MADE languages.

- A method for formalizing clinical guidelines using the MADE languages was presented in Ch. 8. It comprises 7 general rules for identifying MADE archetypes and MADE processes in a clinical guideline, and it was applied to formalize a complete clinical guideline (for gestational diabetes mellitus). Although the guideline was presented using semi-formal workflows instead of natural language English, these rules can be generalised to directly formalise narrative guidelines. The main prerequisite is that the guideline should be readily divided into measurement tasks, decision criteria and therapeutic plans, as these are the main components of the semi-formal workflows.

9.4 Future Directions

9.4.1 Evaluation of the MADE Languages

The MADE languages were validated using a complete clinical guideline for gestational diabetes mellitus, which was developed by a team of expert clinicians and was demonstrated to be clinically relevant in patient trials of the MobiGuide project. However, since the MobiGuide system comprises a fixed number (viz. 2) of knowledge-based systems, the main premise of this thesis, which is to support an arbitrary distribution of knowledge, remains to be evaluated for clinical relevance. Therefore, in collaboration with clinicians, patients and other stakeholders, the MADE languages and their clinical value should be fully evaluated by designing, implementing and deploying "n-ary" guideline-based pervasive healthcare systems for GDM and other clinical applications.

9.4.2 Improvements to the MADE Languages

By fully evaluating the MADE languages, extensions to them can also be identified for future implementation. For example, it has already been established that the MADE guideline language does not currently support partial specification of processes, which may be useful to formalize tasks that must be manually performed and therefore not fully detailed in the clinical guideline. Personalization of guideline knowledge is also not supported; to account for individual differences in, e.g. personal habits and drug dosage requirements, the guideline knowledge must be directly accessed and modified by the clinician before execution. Finally, the MADE languages rely heavily on mathematical expressions to capture the data and processing requirements of clinical guidelines. While fully expressive, these expressions may not be easily interpreted by clinicians and other stakeholders, thus there may be a need to raise the level of abstraction of the MADE languages.

9.4.3 Development of a Knowledge Acquisition Tool

In this thesis, work has been conducted on the application of Rosette to verify clinical guidelines that have been formalized using the MADE languages. In the future, this work can be extended into a full knowledge acquisition tool, which, for example, can include a graphical user interface to formalize clinical guidelines, to develop and execute test data as well as to visualize the test and verification results. Furthermore, formal verification of complete MADE networks instead of individual MADE processes should also be investigated, such as to ensure that mutually exclusive MADE processes would never be activated at the same time.

9.5 Future Outlook

The continual development of computing technologies, such as in artificial intelligence, has enabled and is expected to enable the rise of increasingly intelligent pervasive healthcare systems for an increasing range of applications. For example, Yu et al. in 2021 investigated the use of deep neural networks to predict the health and wellbeing of shift workers [93], while in that same year Prabhu, O'Connor and Kieran presented a deep learning model for detecting and counting exercise

repetition [59]. Although these examples focus of on physical health, growing research has also been and is expected to be conducted on mental health; an example of such research by Motalebi and Abdullah involved the use of Amazon Alexa to deliver cognitive-behavioural conjoint therapy to patients with post-traumatic stress disorder [50].

As pervasive healthcare systems become more prevalent, there is also increasing research on methods to not only enable patient empowerment but also encourage it. For example, Smirnova, Eriksson and Fagerstrøm in 2021 presented thirteen factors that can affect the initial uptake and continual use of mobile health applications, including the availability of measurable outcomes as well as the affordability and flexibility of the application [74]. Furthermore, in the context of gamification, Sienel, Münster and Zimmermann examined how different models of game players may be used to personalise fitness apps with different gaming elements [73].

As implied by these examples, research into pervasive healthcare systems has generally been divided along clear disparate lines, each one focusing on a particular technology, application or concern in general. However, as the the field continues to mature, it is expected that growing synergies will be established between these different research lines. The MADE models presented in this thesis may, for example, be used as a basis for machine learning algorithms, while the outputs of MADE processes may guide the personalisation of healthcare-related games.

References

[1] American Diabetes Association (2014). Diagnosis and Classification of Diabetes Mellitus. *Diabetes Care*, 37(Supplement 1):S81–S90.

[2] Andreu-Perez, J., Leff, D. R., Ip, H. M. D., and Yang, G.-Z. (2015). From Wearable Sensors to Smart Implants–Toward Pervasive and Personalized Healthcare. *IEEE transactions on bio-medical engineering*, 62(12):2750—2762.

[3] Bardram, J. E. (2008). Pervasive Healthcare as a Scientific Discipline. *Methods of Information in Medicine*, 47:178–185.

[4] Beale, T. (2002). Archetypes: Constraint-based Domain Models for Future-proof Information Systems. In *Eleventh OOPSLA Workshop on Behavioral Semantics: Serving the Customer*, pages 16–32.

[5] Boaz, D. and Shahar, Y. (2003). Idan: A Distributed Temporal-Abstraction Mediator for Medical Databases. In Dojat, M., Keravnou, E. T., and Barahona, P., editors, *Artificial Intelligence in Medicine*, pages 21–30, Berlin, Heidelberg. Springer Berlin Heidelberg.

[6] Boaz, D. and Shahar, Y. (2005). A framework for distributed mediation of temporal-abstraction queries to clinical databases. *Artificial Intelligence in Medicine*, 34:3–24.

[7] Boxwala, A. A., Peleg, M., Tu, S., Ogunyemi, O., Zeng, Q. T., Wang, D., Patel, V. L., Greenes, R. A., and Shortliffe, E. H. (2004). GLIF3: a representation format for sharable computer-interpretable clinical practice guidelines. *Journal of Biomedical Informatics*, 37(3):147 – 161.

[8] Brachman, R. and Levesque, H. (2004). *Knowledge Representation and Reasoning*. Morgan Kaufmann.

[9] Bradley, A. R. (2012). Understanding IC3. In Cimatti, A. and Sebastiani, R., editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 1–14, Berlin, Heidelberg. Springer Berlin Heidelberg.

[10] Brambilla, M., Cabot, J., and Wimmer, M. (2017). *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2nd edition.

[11] Bunnin, N. and Yu, J. (2004). *The Blackwell Dictionary of Western Philosophy*. Blackwell Publishing.

[12] Carson, E. R., Cramp, D. G., Morgan, A., and Roudsari, A. V. (1998). Clinical Decision Support, Systems Methodology, and Telemedicine: Their Role in the Management of Chronic Disease. *IEEE Transactions on Information Technology in Biomedicine*, 2:80–88.

[13] Casati, R. and Varzi, A. (2015). Events. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2015 edition.

[14] Ciancarini, P. and Wooldridge, M. J., editors (2001). *Agent-Oriented Software Engineering*. Springer-Verlag Berlin Heidelberg.

[15] Clinical Decision Support Work Group HL7 (2010). *GELLO: A Common Expression Language*, Release 2 edition. Available at http://hl7.ihelse.net/hl7v3/infrastructure/gello/gello.html. Last accessed Apr. 03, 2020.

[16] Clinical Interoperability Council Work Group HL7 (2011). *HL7 Version 3 Standard: Emergency Medical Services Domain Information Model*, Release 1 edition. Available at http://www.hl7.org/v3ballotarchive/v3ballot/html/domains/uvem/uvem.html. Last accessed Mar. 29, 2020.

[17] Cvach, M. (2012). Monitor Alarm Fatigue: An Integrative Review. *Biomedical Instrumentation & Technology*, 46(4):268–277. PMID: 22839984.

[18] de Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

[19] Fitzgerald, J. and Larsen, P. G. (2009). *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, USA, 2nd edition.

[20] Florence, S., Fetscher, B., Flatt, M., Temps, W., Kiguradze, T., West, D., Niznik, C., Yarnold, P., Findler, R., and Belknap, S. (2015). POP-PL: A patient-oriented prescription programming language. In Kastner, C. and Gokhale, A., editors, *GPCE 2015 - Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming*, pages 131–140. Association for Computing Machinery, Inc.

[21] Fung, N. L. S., Jones, V. M., and Hermens, H. J. (2017). The MADE reference information model for interoperable pervasive telemedicine systems. *Methods of Information in Medicine*, 56(2):180–186.

[22] García-Sáez, G., Rigla, M., Martínez-Sarriegui, I. n., Shalom, E., Peleg, M., Broens, T., Pons, B., Caballero-Ruíz, E., Gómez, E. J., and Hernando, M. E. (2014). Patient-oriented Computerized Clinical Guidelines for Mobile Decision Support in Gestational Diabetes. *Journal of Diabetes Science and Technology*, 8:238–246.

[23] Giordano, L., Terenziani, P., Bottrighi, A., Montani, S., and Donzella, L. (2006). Model checking for clinical guidelines: an agent-based approach. In *AMIA Annual Symposium Proceedings*, pages 289–293.

[24] Gogolla, M., B uttner, F., and Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1):27 – 34. Special issue on Experimental Software and Toolkits.

[25] Goldstein, A., Quaglini, S., Sacchi, L., Panzarasa, S., Napolitano, C., Larburu, N., Bults, R., Rigla, M., and García-Sáez, G. (2014). Deliverable 4.1: CIGs KB. Technical report, MobiGuide Project (FP7-287811). Retrieved May 08, 2015, from http://www.mobiguide-project.eu/downloads/category/10-public-deliverables.

[26] González-Ferrer, A., Peleg, M., Parimbelli, E., Shalom, E., Marcos, C., Klebanov, G., Martinez-Sarriegui, I., Fung, N. L. S., and Broens, T. (2014). Use of the virtual medical record data model for communication among components of a distributed decision-support system. In *2014 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*, pages 526–530.

[27] Goud, R., Hasman, A., and Peek, N. (2008). Development of a guideline-based decision support system with explanation facilities for outpatient therapy. *Computer Methods and Programs in Biomedicine*, 91(2):145–153.

[28] Habib, M., Mohktar, M., Kamaruzzaman, S., Lim, K., Pin, T., and Ibrahim, F. (2014). Smartphone-Based Solutions for Fall Detection and Prevention: Challenges and Open Issues. *Sensors*, 14(4):7181–7208.

[29] Hatsek, A., Shahar, Y., Taieb-Maimon, M., Shalom, E., Klimov, D., and Lunenfeld, E. (2010). A Scalable Architecture for Incremental Specification and Maintenance of Procedural and Declarative Clinical Decision-Support Knowledge. *The Open Medical Informatics Journal*, 4:255–277.

[30] Huzooree, G., Khedo, K. K., and Joonas, N. (2019). Pervasive mobile healthcare systems for chronic disease monitoring. *Health Informatics Journal*, 25(2):267–291. PMID: 28464728.

[31] Institute of Medicine (2011). *Clinical Practice Guidelines We Can Trust*. The National Academies Press, Washington, DC.

[32] International, H. (2011). *Template*. Available at https://openehr.atlassian.net/wiki/spaces/healthmod/pages/2949191/Introduction+to+Archetypes+and+Archetype+classes. Last accessed Apr. 20, 2020.

[33] Isern, D. and Moreno, A. (2008). Computer-based execution of clinical guidelines: A review. *International Journal of Medical Informatics*, 77:787–808.

[34] ISO/IEC (1996). Information technology – Syntactic metalanguage – Extended BNF. Technical Report 14977 : 1996(E), ISO/IEC.

[35] Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[36] Johnson, P., Tu, S., Booth, N., Sugden, B., and Purves, I. (2000). Using scenarios in chronic disease management guidelines for primary care. *AMIA Annual Symposium Proceedings*, pages 389–93.

[37] Kawamoto, K., Del Fiol, G., Strasberg, H., Hulse, N., Curtis, C., Cimino, J., Rocha, B., Maviglia, S., Fry, E., Scherpbier, H., Huser, V., Redington, P., Vawdrey, D., Dufour, J.-C., Price, M., Weber, J., White, T., Hughes, K., Mcclay, J., and McIntyre, A. (2010). Multi-National, Multi-Institutional Analysis of Clinical Decision Support Data Needs to Inform Development of the HL7 Virtual Medical Record Standard. *AMIA Annual Symposium Proceedings*, 2010:377–81.

[38] Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer*, 36:41–50.

[39] Klaassen, B., van Beijnum, B.-J., Weusthof, M., Hofs, D., van Meulen, F., Luinge, H., Lorussi, F., Hermens, H., and Veltink, P. (2014). A System for Monitoring Stroke Patients in a Home Environment. In *International Conference on Health Informatics (HEALTHINF 2014)*, pages 125–132.

[40] Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., USA.

[41] Klimov, D. and Shahar, Y. (2013). iALARM: An Intelligent Alert Language for Activation, Response, and Monitoring of Medical Alerts. In Riaño, D., Lenz, R., Miksch, S., Peleg, M., Reichert, M., and ten Teije, A., editors, *Process Support and Knowledge Representation in Health Care*, pages 128–142, Cham. Springer International Publishing.

[42] Larburu, N., Bults, R., van Sinderen, M., Widya, I., and Hermens, H. (2015). An Ontology for Telemedicine Systems Resiliency to Technological Context Variations in Pervasive Healthcare. *IEEE Journal of Translational Engineering in Health and Medicine*, 3:1–10.

[43] Lasierra, N., Alesanco, A., and García, J. (2014). Designing an Architecture for Monitoring Patients at Home: Ontologies and Web Services for Clinical and Technical Management Integration. *IEEE Journal of Biomedical and Health Informatics*, 18:896–906.

[44] Leslie, H. (2012). Introduction to Archetypes and Archetype Classes. Technical report, openEHR. Available at https://openehr.atlassian.net/wiki/spaces/healthmod/pages/2949191/Introduction+to+Archetypes+and+Archetype+classes. Last accessed Apr. 20, 2020.

[45] Marcos, C., González-Ferrer, A., Peleg, M., and Cavero, C. (2015). Solving the interoperability challenge of a distributed complex patient guidance system: a data integrator based on HL7's Virtual Medical Record standard. *Journal of the American Medical Informatics Association*, 22:587–599.

[46] Mayo Clinic (2020). *Gestational Diabetes*. Available at https://www.mayoclinic.org/diseases-conditions/gestational-diabetes/symptoms-causes/syc-20355339. Last accessed Apr. 30, 2020.

[47] Michalowski, M., Wilk, S., Michalowski, W., and Carrier, M. (2019). Mitplan: A planning approach to mitigating concurrently applied clinical practice guidelines. In Riaño, D., Wilk, S., and ten Teije, A., editors, *Artificial Intelligence in Medicine*, pages 93–103. Springer International Publishing.

[48] Ming, W.-K., Mackillop, L. H., Farmer, A. J., Loerup, L., Bartlett, K., Levy, J. C., Tarassenko, L., Velardo, C., Kenworthy, Y., and Hirst, J. E. (2016). Telemedicine Technologies for Diabetes in Pregnancy: A Systematic Review and Meta-Analysis. *Journal of Medical Internet Research*, 18(11):e290.

[49] Monostori, L., Váncza, J., and Kumara, S. (2006). Agent-Based Systems for Manufacturing. *CIRP Annals*, 55(2):697 – 720.

[50] Motalebi, N. and Abdullah, S. (2018). Conversational agents to provide couple therapy for patients with ptsd. In *Proceedings of the 12th EAI International Conference on Pervasive Computing Technologies for Healthcare*, PervasiveHealth '18, page 347–351, New York, NY, USA. Association for Computing Machinery.

[51] National Health Service (NHS), UK (2012). Cold or flu? Retrieved March 31, 2014, from http://www.nhs.uk/Livewell/coldsandflu/Pages/Isitacoldorflu.aspx.

[52] National Institute for Health and Clinical Excellence (NICE), UK (2006). Obesity: Guidance on the prevention, identification, assessment and management of overweight and obesity in adults and children. Retrieved November 27, 2014, from http://www.nice.org.uk/guidance/cg43.

[53] Object Management Group (2006). *MOF Model to Text Transformation Language*, v1.0 edition. Available at https://www.omg.org/spec/MOFM2T/1.0/PDF. Last accessed May. 03, 2020.

[54] Object Management Group (2017). *OMG Unified Modeling Language (OMG UML)*, Version 2.5.1 edition. Available at https://www.omg.org/spec/UML/2.5.1/PDF. Last accessed May. 03, 2020.

[55] Peleg, M., Keren, S., and Denekamp, Y. (2008). Mapping computerized clinical guidelines to electronic medical records: Knowledge-data ontological mapper (kdom). *Journal of Biomedical Informatics*, 41(1):180–201.

[56] Peleg, M., Shahar, Y., Quaglini, S., Broens, T., Budasu, R., Fung, N., Fux, A., García-Sáez, G., Goldstein, A., González-Ferrer, A., Hermens, H., Hernando, M. E., Jones, V., Klebanov, G., Klimov, D., Knoppel, D., Larburu, N., Marcos, C., Martínez-Sarriegui, I., Napolitano, C., Àngels Pallàs, Palomares, A., Parimbelli, E., Pons, B., Rigla, M., Sacchi, L., Shalom, E., Soffer, P., and van Schooten, B. (2017a). Assessment of a personalized and distributed patient guidance system. *International Journal of Medical Informatics*, 101:108–130.

[57] Peleg, M., Shahar, Y., Quaglini, S., Fux, A., García-Sáez, G., Goldstein, A., Hernando, M. E., Klimov, D., Martínez-Sarriegui, I. n., Napolitano, C., and et al. (2017b). MobiGuide: A Personalized and Patient-Centric Decision-Support System and Its Evaluation in the Atrial Fibrillation and Gestational Diabetes Domains. *User Modeling and User-Adapted Interaction*, 27(2):159–213.

[58] Peleg, M., Tu, S., Bury, J., Ciccarese, P., Fox, J., Greenes, R. A., Hall, R., Johnson, P. D., Jones, N., Kumar, A., Miksch, S., Quaglini, S., Seyfang, A., Shortliffe, E. H., and Stefanelli, M. (2003). Comparing Computer-interpretable Guideline Models: A Case-study Approach. *Journal of the American Medical Informatics Association*, 10:52–68.

[59] Prabhu, G., O'Connor, N. E., and Moran, K. (2021). A deep learning model for exercise-based rehabilitation using multi-channel time-series data from a single wearable sensor. In Ye, J., O'Grady, M. J., Civitarese, G., and Yordanova, K., editors, *Wireless Mobile Communication and Healthcare*, pages 104–115, Cham. Springer International Publishing.

[60] Qi, J., Yang, P., Min, G., Amft, O., Dong, F., and Xu, L. (2017). Advanced internet of things for personalised healthcare systems: A survey. *Pervasive and Mobile Computing*, 41:132 – 149.

[61] Rigla, M., Tirado, R., Caixàs, A., Pons, B., and Costa, J. (2013). Gestational Diabetes Guideline CSPT. Technical report, MobiGuide Project (FP7-287811). Version 1.0, 12/02/2013. Internal document.

[62] Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., 3 edition.

[63] Samwald, M., Fehre, K., de Bruin, J., and Adlassnig, K.-P. (2012). The Arden Syntax standard for clinical decision support: Experiences and directions. *Journal of Biomedical Informatics*, 45(4):711 – 718.

[64] Sargious, P. (2007). *Chronic Disease Prevention and Management*. Health Canada. Available at www.healthcanada.gc.ca/phctf. Last accessed May. 05, 2020.

[65] Seyfang, A., Kosara, R., and Miksch, S. (2002a). *Asbru Reference Manual*, 2 edition. Retrieved May 08, 2015, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.200.4550.

[66] Seyfang, A., Miksch, S., and Marcos, M. (2002b). Combining diagnosis and treatment using ASBRU. *International Journal of Medical Informatics*, 68:49–57.

[67] Shahar, Y. (1997). A framework for knowledge-based temporal abstraction. *Artificial Intelligence*, 90:79–133.

[68] Shahar, Y., Miksch, S., and Johnson, P. (1996). An intention-based language for representing clinical guidelines. In *Proceedings of the AMIA Annual Fall Symposium*, pages 592–596.

[69] Shahar, Y., Young, O., Shalom, E., Galperin, M., Mayaffit, A., Moskovitch, R., and Hessing, A. (2004). A framework for a distributed, hybrid, multiple-ontology clinical-guideline library, and automated guideline-support tools. *Journal of Biomedical Informatics*, 37:325–344.

[70] Shalom, E., Shahar, Y., Goldstein, A., Ariel, E., Sheinberger, M., Fung, N., Jones, V., and van Schooten, B. (2015). Implementation of a Distributed Guideline-Based Decision Support Model Within a Patient-Guidance Framework. In Riaño, D., Lenz, R., Miksch, S., Peleg, M., Reichert, M., and ten Teije, A., editors, *Knowledge Representation for Health Care*, pages 111–125. Springer International Publishing.

[71] Shalom, Erez fand Shahar, Y., Taieb-Maimon, M., Bar, G., Yarkoni, A., Young, O., Martins, S. B., Vaszar, L., Goldstein, M. K., Liel, Y., Leibowitz, A., Marom, T., and Lunenfeld, E. (2008). A quantitative assessment of a methodology for collaborative specification and evaluation of clinical guidelines. *Journal of Biomedical Informatics*, 41:889–903.

[72] Shortliffe, E. H. (1976). *Computer-Based Medical Consultations: MYCIN*. American Elsevier Publishing Co., Inc., USA.

[73] Sienel, N., Münster, P., and Zimmermann, G. (2021). Player-type-based personalization of gamification in fitness apps. In *Proceedings of the 14th International Joint Conference on Biomedical Engineering Systems and Technologies - HEALTHINF,*, pages 361–368. INSTICC, SciTePress.

[74] Smirnova, E., Eriksson, N., and Fagerstrøm, A. (2021). Adoption and use of health-related mobile applications: A qualitative study with experienced users. In *Proceedings of the 14th International Joint Conference on Biomedical Engineering Systems and Technologies - HEALTHINF,*, pages 288–295. INSTICC, SciTePress.

[75] Sommerville, I. (2016). *Software Engineering, 10th Edition*. Pearson.

[76] Sutton, D. R. and Fox, J. (2003). The Syntax and Semantics of the PRO*forma* Guideline Modeling Language. *Journal of the American Medical Informatics Association*, 10:433–443.

[77] Tal, E. (2017). Measurement in Science. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2017 edition.

[78] Taylor, R. N., Medvidović, N., and Dashofy, E. M. (2010). *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Inc.

[79] ten Teije, A., Marcos, M., Balser, M., van Croonenborg, J., Duelli, C., van Harmelen, F., Lucas, P., Miksch, S., Reif, W., Rosenbrand, K., and Seyfang, A. (2006). Improving medical protocols by formal methods. *Artificial Intelligence in Medicine*, 36(3):193–209.

[80] Terenziani, P., Molino, G., and Torchio, M. (2001). A modular approach for representing and executing clinical guidelines. *Artificial Intelligence in Medicine*, 23(3):249–276.

[81] The openEHR Foundation (2016). *Archetype Definition Language 2 (ADL2)*, AM Release 2.0.6 edition. Available at https://specifications.openehr.org/releases/AM/Release-2.0.6/ADL2.html. Last accessed Apr. 20, 2020.

[82] The openEHR Foundation (2018a). *EHR Information Model*, RM Release 1.0.4 edition. Available at https://specifications.openehr.org/releases/RM/Release-1.0.4/ehr.html. Last accessed Mar. 29, 2020.

[83] The openEHR Foundation (2018b). *OpenEHR Architecture Overview*, BASE Release 1.1.0 edition. Available at https://specifications.openehr.org/releases/BASE/Release-1.1.0/architecture_overview.html. Last accessed Mar. 29, 2020.

[84] The openEHR Foundation (2019). *Guideline Definition Language v2 (GDL2)*, CDS Release 2.0.0 edition. Available at https://specifications.openehr.org/releases/CDS/latest/GDL2.html. Last accessed Mar. 29, 2020.

[85] Torlak, E. and Bodik, R. (2013). Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 135–152, New York, NY, USA. Association for Computing Machinery.

[86] Varshney, U. (2009). *Pervasive Healthcare Computing: EMR/EHR, Wireless and Health Monitoring*. Springer Publishing Company, Incorporated, 1st edition.

[87] Wagner, E. H. (1998). Chronic disease management: what will it take to improve care for chronic illness? *Effective clinical practice : ECP*, 1:2–4.

[88] Wilk, S., Michalowski, W., Michalowski, M., Farion, K., Hing, M. M., and Mohapatra, S. (2013). Mitigation of adverse interactions in pairs of clinical practice guidelines using constraint logic programming. *Journal of Biomedical Informatics*, 46(2):341–353.

[89] Wilson, G. and Shpall, S. (2016). Action. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2016 edition.

[90] Woods, W. A. (1986). Important issues in knowledge representation. *Proceedings of the IEEE*, 74(10):1322–1334.

[91] World Health Organization (2018). *Noncommunicable Diseases Country Profiles 2018*.

[92] Young, O. and Shahar, Y. (2005). The Spock System: Developing a Runtime Application Engine for Hybrid-Asbru Guidelines. In Miksch, S., Hunter, J., and Keravnou, E. T., editors, *Artificial Intelligence in Medicine*, pages 166–170, Berlin, Heidelberg. Springer Berlin Heidelberg.

[93] Yu, H., Itoh, A., Sakamoto, R., Shimaoka, M., and Sano, A. (2021). Forecasting health and wellbeing for shift workers using job-role based deep neural network. In Ye, J., O'Grady, M. J., Civitarese, G., and Yordanova, K., editors, *Wireless Mobile Communication and Healthcare*, pages 89–103, Cham. Springer International Publishing.

[94] Zanni, C., Goc, M. L., and Frydman, C. (2006). A Conceptual Framework for the Analysis, Classification and Choice of Knowledge-Based Diagnosis Systems. *International Journal of Knowledge-based and Intelligent Engineering Systems*, 10(2):113–138.

[95] Zhong, A., Choudhary, P., McMahon, C., Agrawal, P., Welsh, J., Cordero, T., and Kaufman, F. (2016). Effectiveness of Automated Insulin Management Features of the MiniMed® 640G Sensor-Augmented Insulin Pump. *Diabetes Technology and Therapeutics*, 18(10):657–663.

Appendix A

Test Cases for the MADE Archetypes

The following is the source code (re-formatted to fit the page size) for all the 17 archetypes used to verify the implementation of the MADE RIM. This source code is also available at https://github.com/nlsfung/MADE-Language/blob/master/exp/TestArchetypes.rkt, and the details of the MADE RIM can be found in Ch. 4.

```
#lang rosette/safe

(require "../lang/VerifySyntax.rkt"
         "../lang/IMSyntax.rkt"
         "../lang/NomEnumSyntax.rkt"
         "../rim/BasicDataTypes.rkt")

(provide (all-defined-out))

; This file contains the specification of the archetypes used to
; test the implementation of the MADE archetype language. Please
; refer to Table 5.2 and Sec. 5.4 of the PhD thesis for more
; details about the tests. The specific; outputs of each test
; are documented as comments in this specification.
```

```

; > (verify-archetype arch-01)
; Example #<procedure:arch-01>:
; #(struct:arch-01 #f #(struct:datetime 2019 12 15 0 0 0)
;   #(struct:dimensioned dim$0 units))
(define-measurement arch-01 'units)

; > (verify-archetype arch-02)
; Example #<procedure:arch-02>:
; (unsat)
(define-measurement arch-02 'units (lambda (v) #f))

; > (verify-archetype arch-03)
; Example #<procedure:arch-03>:
; #(struct:arch-03 #f #(struct:datetime 2019 12 15 0 0 0)
;   #(struct:arch-03-value-space a))
(define-observation arch-03 nominal 'a 'b 'c)

; > (verify-archetype arch-04)
; Example #<procedure:arch-04>:
; #(struct:arch-04 #f #(struct:datetime 2019 12 15 0 0 0)
;   #(struct:count 0))
(define-observation arch-04 count)

; > (verify-archetype arch-05)
; Example #<procedure:arch-05>:
; (unsat)
(define-observation arch-05 count
  (lambda (d) (and (> (get-value d) 50) (< (get-value d) 10))))

; > (verify-archetype arch-06)
; Example #<procedure:arch-06>:
; #(struct:arch-06 #f #(struct:datetime-range
```

```
;   #(struct:datetime 2019 12 15 1 0 0)
;   #(struct:datetime 2019 12 15 23 0 0))
;   #(struct:bool bool-val$0))
(define-observation arch-06 #:event)

; > (verify-archetype arch-07)
; Example #<procedure:arch-07>:
; #(struct:arch-07 #f #(struct:datetime-range
;   #(struct:datetime 2019 12 15 1 0 0)
;   #(struct:datetime 2019 12 15 1 0 0))
;   #(struct:arch-07-value-space b))
(define-abstraction arch-07 enumerated 'a 'b)

; > (verify-archetype arch-08)
; Example #<procedure:arch-08>:
; #(struct:arch-08 #f #(struct:datetime-range
;   #(struct:datetime 2019 12 15 1 0 0)
;   #(struct:datetime 2019 12 15 1 0 0))
;   #(struct:proportion 11))
(define-abstraction arch-08 proportion
  (lambda (d) (> (get-value d) 10)))

; > (verify-archetype arch-09)
; Example #<procedure:arch-09>:
; (unsat)
(define-abstraction arch-09 proportion
  (lambda (d) (eq? (get-value d) #f)))

; > (verify-archetype arch-10)
; Example #<procedure:arch-10>:
; #(struct:arch-10 #f #(struct:datetime 2019 12 15 0 0 0)
;   (#(struct:scheduled-homogeneous-action arch-13
;    #(struct:schedule (#(struct:datetime 2019 12 15 0 0 0))
```

```
;       {2773771508841512397:2})
;       #(struct:dimensioned dim$0 units)
;    #(struct:duration dur-part$0 dur-part$1 dur-part$2
;       dur-part$3))))
(define-action-plan arch-10 (homogeneous-action 'arch-13))

; > (verify-archetype arch-11)
; Example #<procedure:arch-11>:
; #(struct:arch-11 #f #(struct:datetime 2019 12 15 0 0 0)
;   (#(struct:scheduled-culminating-action arch-15
;    #(struct:schedule (#(struct:datetime 2019 12 15 0 0 0))
;       {2128385024956610424:2}) #(struct:count 0))))
(define-action-plan arch-11 (culminating-action 'arch-15))

; > (verify-archetype arch-12)
; Example #<procedure:arch-12>:
; #(struct:arch-12 #f #(struct:datetime 2019 12 15 0 0 0)
;   (#(struct:scheduled-control proc-1 #(struct:schedule
;     (#(struct:datetime 2019 12 15 0 0 0))
;         {-1411548690416635397:2}) status$0)
;   #(struct:scheduled-culminating-action arch-15
;      #(struct:schedule (#(struct:datetime 2019 12 15 0 0 0))
;          {-2807555162316629719:2}) #(struct:count 0))))
(define-action-plan arch-12 (culminating-action 'arch-15)
  (control 'proc-1))

; > (verify-archetype arch-13)
; Example #<procedure:arch-13>:
; #(struct:arch-13 #f #(struct:datetime 2019 12 15 0 0 0)
;   #(struct:dimensioned dim$0 units)
;   #(struct:duration dur-part$8 dur-part$9
;       dur-part$10 dur-part$11))
(define-action-instruction arch-13 #:homogeneous 'units)
```

```

; > (verify-archetype arch-14)
; Example #<procedure:arch-14>:
; (unsat)
(define-action-instruction arch-14 #:homogeneous 'units
  (lambda (r v) #f))

; > (verify-archetype arch-15)
; Example #<procedure:arch-15>:
; #(struct:arch-15 #f #(struct:datetime 2019 12 15 0 0 0)
;   #(struct:count 0))
(define-action-instruction arch-15 #:culminating count)

; > (verify-archetype arch-16)
; Example #<procedure:arch-16>:
; (unsat)
(define-action-instruction arch-16 #:culminating bool
  (lambda (v) (eq? v 1)))

; > (verify-archetype arch-17)
; Example #<procedure:arch-17>:
; #(struct:arch-17 #f proc-1
;   #(struct:datetime 2019 12 15 0 0 0)
;   #(struct:schedule (#(struct:datetime 2019 12 15 0 0 0))
;   {1484878925140550989:2}) #<void>)
(define-control-instruction arch-17 'proc-1 'proc-2)
```

Appendix B

Test Cases for the MADE Processes

The following is the source code used to manually verify the implementation of the data list generator (`generate-list`) and process verifier (`verify-process`) presented in Sec. 7.2.8 and Sec. 7.2.9 respectively. This source code is also available at https://github.com/nlsfung/MADE-Language/blob/master/exp/TestProcesses.rkt, and for conciseness, the test results are not documented as part of the source code.

```
#lang rosette/safe

(require "../lang/IMSyntax.rkt"
         "../lang/PMSyntax.rkt"
         "../lang/VerifySyntax.rkt"
         "../rim/BasicDataTypes.rkt"
         "../rim/TemporalDataTypes.rkt"
         "../rim/MadeDataStructures.rkt"
         "../rpm/MadeProcess.rkt"
         "../rpm/MonitoringProcess.rkt"
         "../rpm/AnalysisProcess.rkt"
         "../rpm/DecisionProcess.rkt"
         "../rpm/EffectuationProcess.rkt")
```

```
14
15 (provide (all-defined-out))
16
17 ; This file contains the tests performed to verify the
18 ; implementation of generate-list and verify-process.
19
20 ; Specification of the relevant MADE archetypes.
21 (define-measurement body-speed 'ms-1)
22
23 (define-observation sprint-event #:event)
24
25 (define-abstraction exercise-abstraction nominal
26   'insufficient 'sufficient 'excessive)
27
28 (define-action-plan exercise-plan
29   (homogeneous-action 'sprint-action)
30   (culminating-action 'endurance-running-action))
31
32 (define-action-instruction sprint-action
33   #:homogeneous 'ms-1)
34
35 (define-action-instruction endurance-running-action
36   #:culminating dimensioned 'm)
37
38 (define-action-instruction treadmill-output
39   #:culminating dimensioned 'm)
40
41 ; Specification of the processes (and associated procedures).
42 (define-syntax-rule (average dSet)
43   (dimensioned
44    (/ (foldl (lambda (d result)
45                (+ (get-value (measurement-value d)) result))
46                  0
```

```
                  dSet)
      5)
   'ms-1))

(define-monitoring #:event monitor-sprint #f sprint-event
  (event-trigger
    (duration 0 0 0 5)
    (lambda (dSet)
      (dim>=? (average (filter (lambda (d) (body-speed? d))
                               dSet))
              (dimensioned 3 'ms-1))))
  (event-trigger
    (duration 0 0 0 5)
    (lambda (dSet)
      (dim<? (average (filter (lambda (d) (body-speed? d))
                              dSet))
             (dimensioned 2 'ms-1)))))

(define-syntax-rule (count-sprints dSet)
  (length (filter (lambda (d)
                    (and (sprint-event? d)
                         (get-value (observed-event-value d))))
                  dSet)))

(define-analysis analyse-exercise #f exercise-abstraction
  ((duration 30 0 0 0)
   (lambda (dSet) (< (count-sprints dSet) 5))
   (lambda (dSet)
     (exercise-abstraction-value-space 'insufficient)))
  ((duration 14 0 0 0)
   (lambda (dSet) (and (> (count-sprints dSet) 10)
                       (< (count-sprints dSet) 15)))
   (lambda (dSet)
```

```
      (exercise-abstraction-value-space 'sufficient)))
  ((duration 7 0 0 0)
   (lambda (dSet) (> (count-sprints dSet) 10))
   (lambda (dSet)
     (exercise-abstraction-value-space 'excessive))))

(define-decision
  decide-exercise #f exercise-plan
  (#:instructions
   (homogeneous-action-template
     'sprint-action
     (relative-schedule #:rounding (duration 1 0 0 0)
                        #:pattern (duration 0 7 0 0)
                        #:interval (duration 14 0 0 0))
     (dimensioned 3 'ms-1)
     (duration 0 0 0 30))
   (culminating-action-template
     'endurance-running-action
     (relative-schedule #:rounding (duration 1 0 0 0)
                        #:pattern (duration 7 17 0 0)
                        #:interval (duration 14 0 0 0))
     (dimensioned 5000 'm)))
  (#:criteria
    (lambda (dSet)
      (findf
        (lambda (d)
          (and (exercise-abstraction? d)
               (eq? (abstraction-value d)
                    (exercise-abstraction-value-space
                      'insufficient))))
        dSet))))

(define-effectuation effectuate-running #f treadmill-output
```

```
  (target-schedule #:plan exercise-plan
                   #:instruction 'endurance-running-action
                   #:predicate (lambda (inst-set) #t)))

; Execution of verify-process and generate-list.
(verify-process  monitor-sprint
                 (generate-list body-speed
                                (datetime 2019 3 10 0 0 0)
                                (datetime 2019 3 10 0 0 59)
                                3)
                 (datetime 2019 3 10 0 1 0))

(verify-process  analyse-exercise
                 (generate-list sprint-event
                                (datetime 2020 3 17 0 0 0)
                                (datetime 2020 3 17 23 0 0)
                                (duration 0 2 0 0))
                 (datetime 2020 3 18 0 0 0))

(verify-process decide-exercise
                (generate-list exercise-abstraction
                               (datetime 2020 3 1 0 0 0)
                               (datetime 2020 3 15 23 0 0)
                               1)
                (datetime 2020 3 15 0 0 0))

(verify-process effectuate-running
                (generate-list exercise-plan
                               (datetime 2020 3 1 0 0 0)
                               (datetime 2020 3 15 23 0 0)
                               1)
                (datetime 2020 3 15 0 0 0))
```

Appendix C

Clinical Guideline for Gestational Diabetes Mellitus

The following pages contain 13 semi-formal workflows that are reproduced from [25] and together constitute a complete clinical guideline for gestational diabetes mellitus (GDM). These workflows have been developed as part of the MobiGuide project to validate the clinical relevance of the MobiGuide system (Sec. 1.4) [22] and are used as inputs in this research to validate the MADE languages and underlying models. Of these 13 workflows, 2 relate to the management of blood glucose (BG), 2 to urinary ketone levels (ketonuria), 1 to diet compliance, 1 to physical activity (PA) compliance and 7 to blood pressure (BP).

C.1 Blood Glucose Monitoring Plan Flow

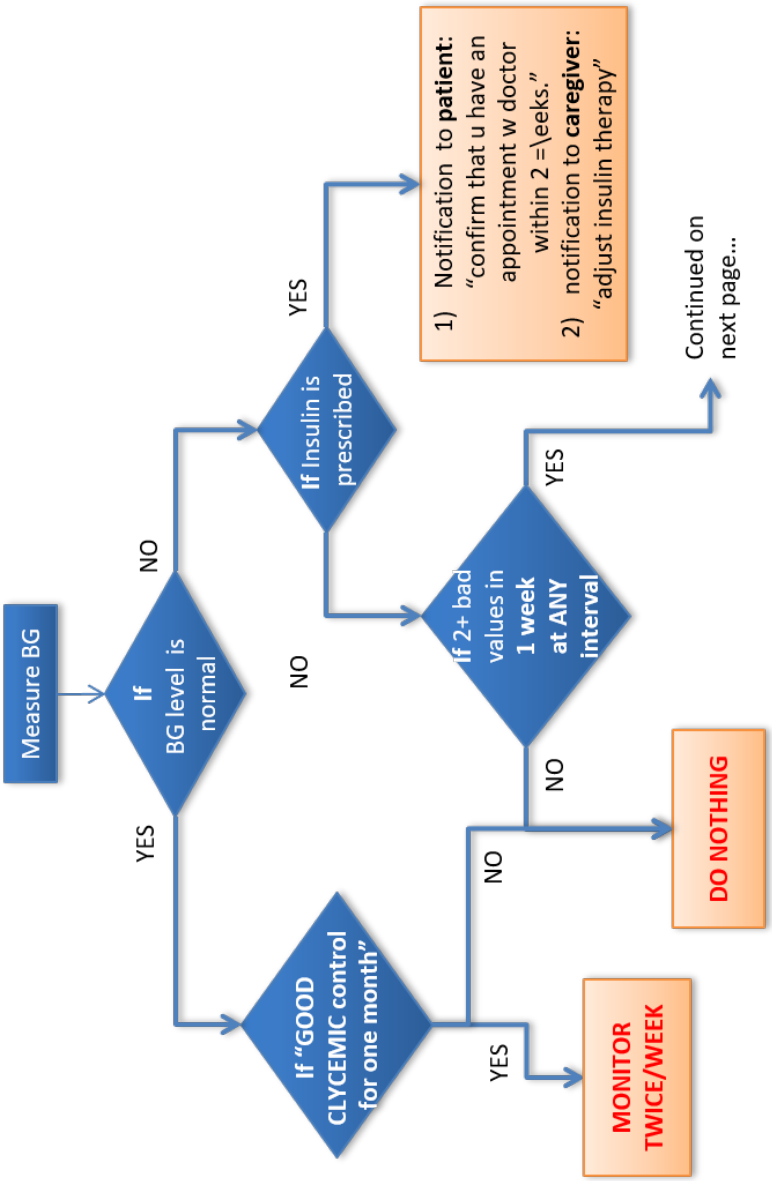

Fig. C.1 Part 1 of 2 of a workflow for monitoring blood glucose four times daily (source: [25]).

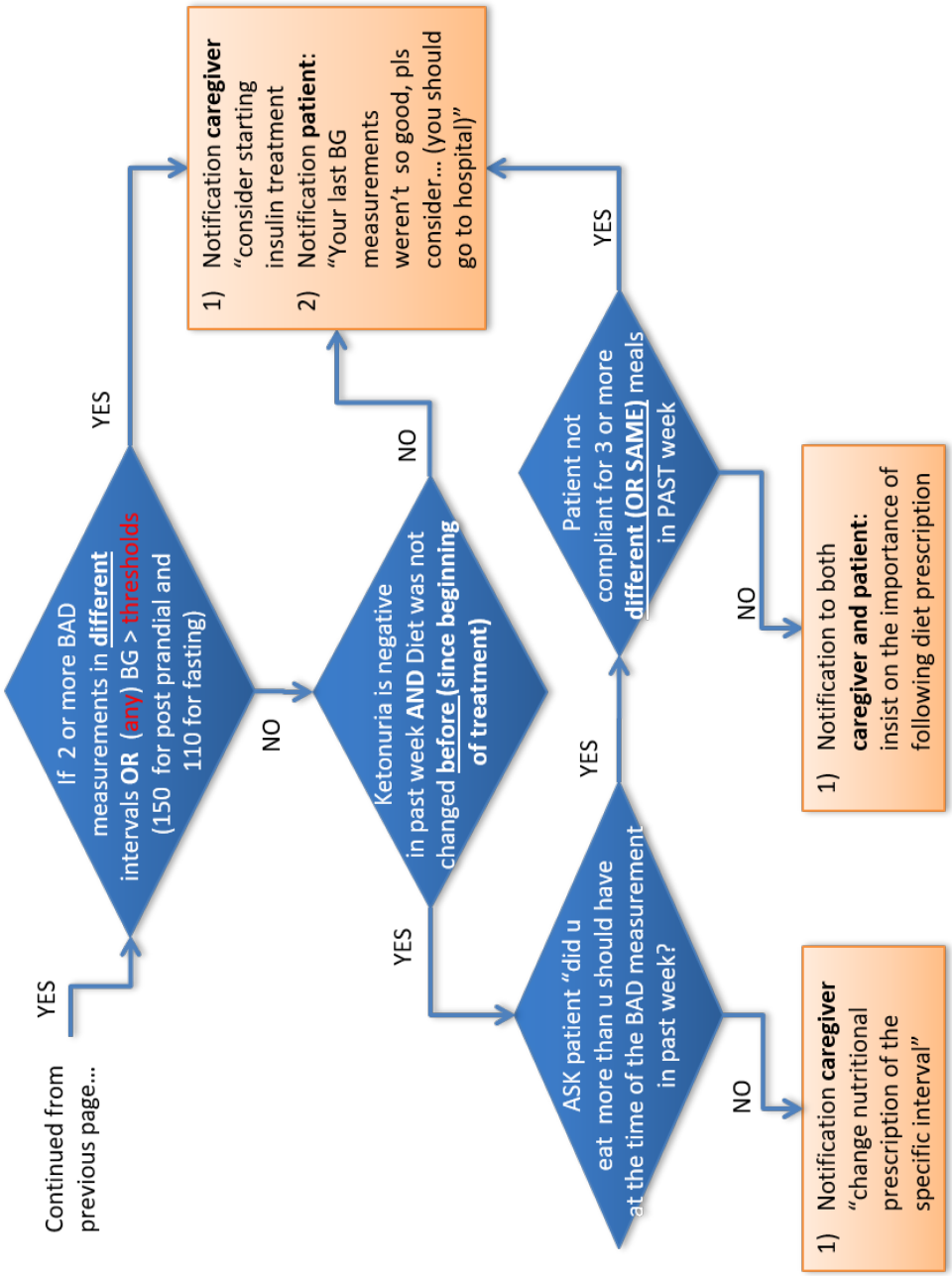

Fig. C.2 Part 2 of 2 of a workflow for monitoring blood glucose four times daily (source: [25]).

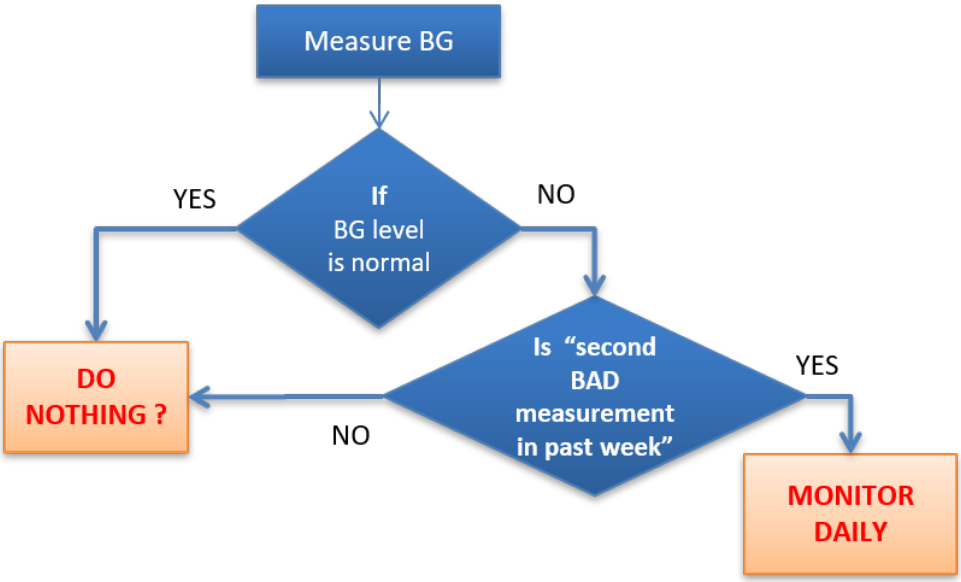

Fig. C.3 Workflow for monitoring blood glucose for two days each week, four times each day (source: [25]).

C.2 Ketonuria Monitoring Plan Flow

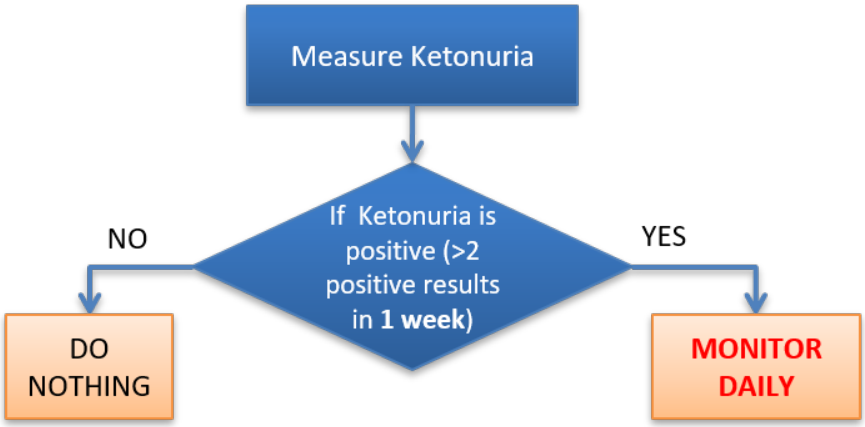

Fig. C.4 Workflow for monitoring ketonuria (urinary ketones) twice weekly (source: [25]).

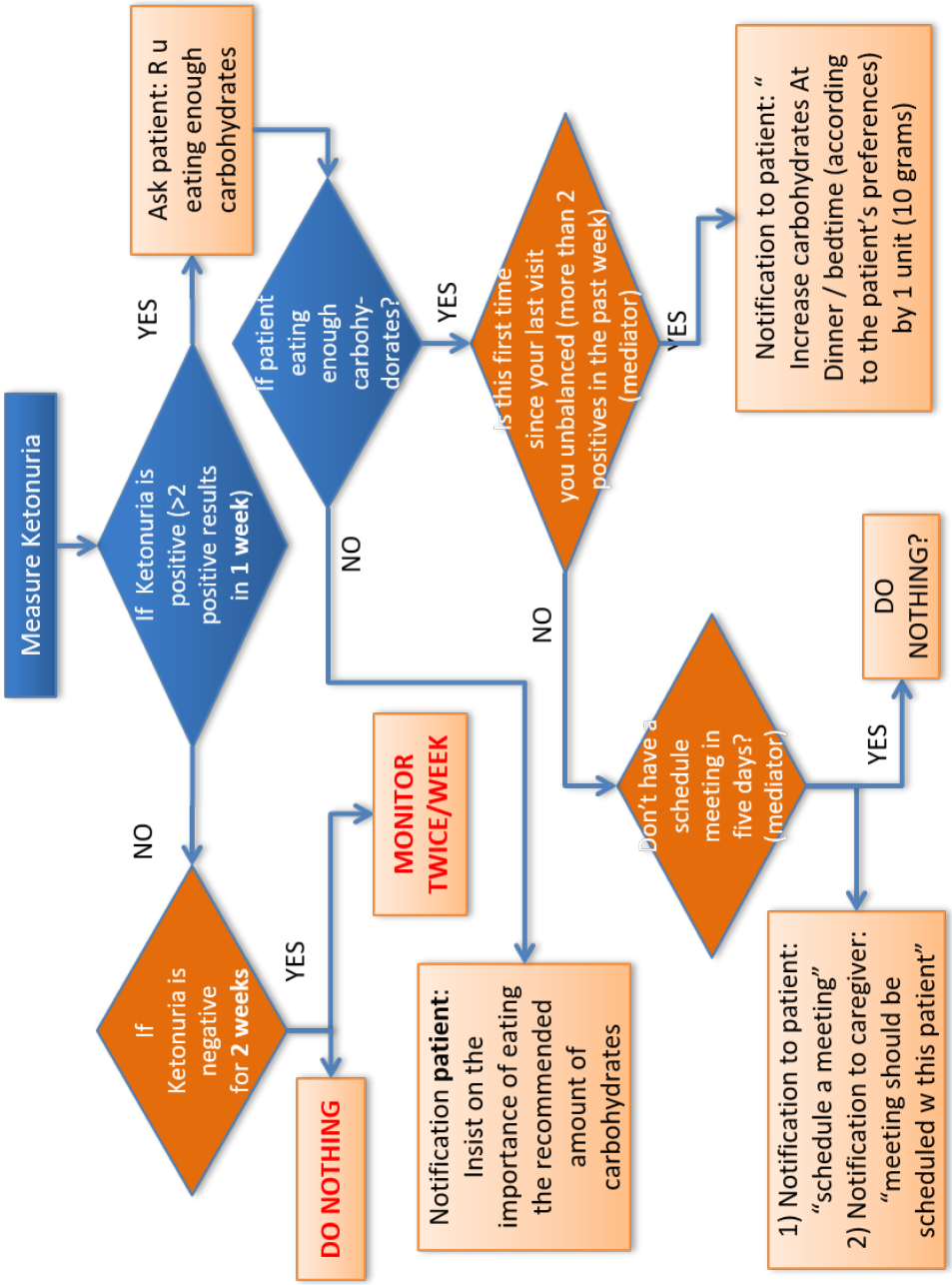

Fig. C.5 Workflow for monitoring ketonuria daily (source: [25]).

C.3 Diet Monitoring Plan Flow

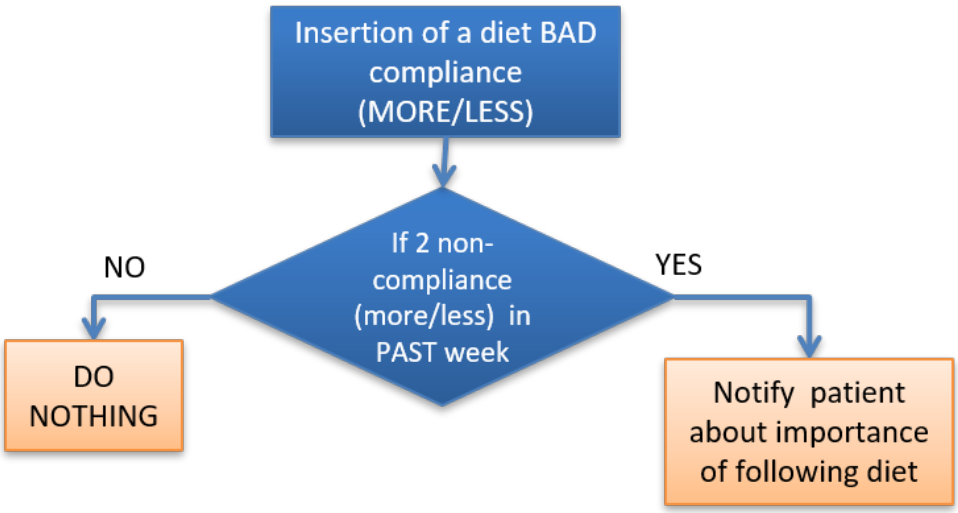

Fig. C.6 Workflow for monitoring diet (source: [25]).

C.4 Exercise Monitoring Plan Flow

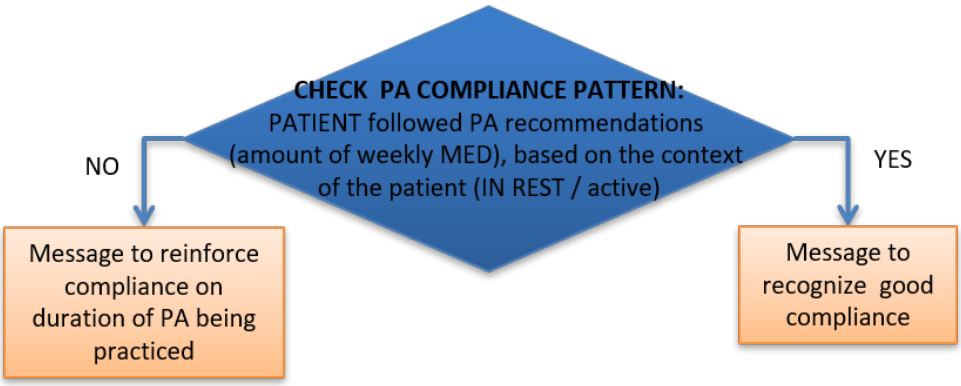

Fig. C.7 Workflow for monitoring exercise once weekly (source: [25]).

C.5 Blood Pressure Monitoring Plan Flow

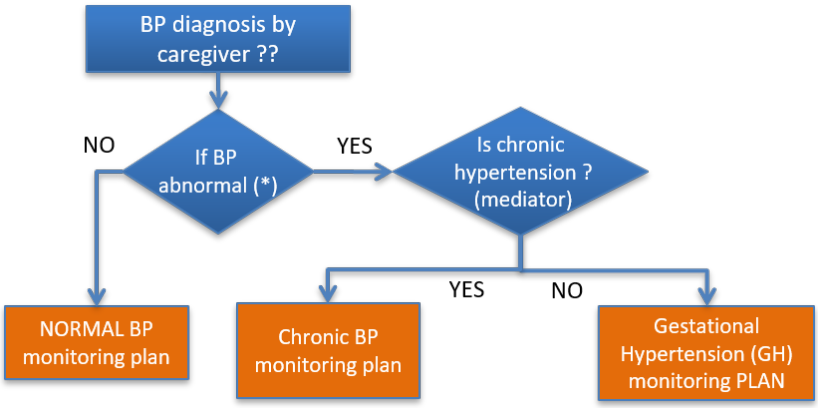

Fig. C.8 Workflow for choosing the initial blood pressure (BP) monitoring plan (source: [25]).

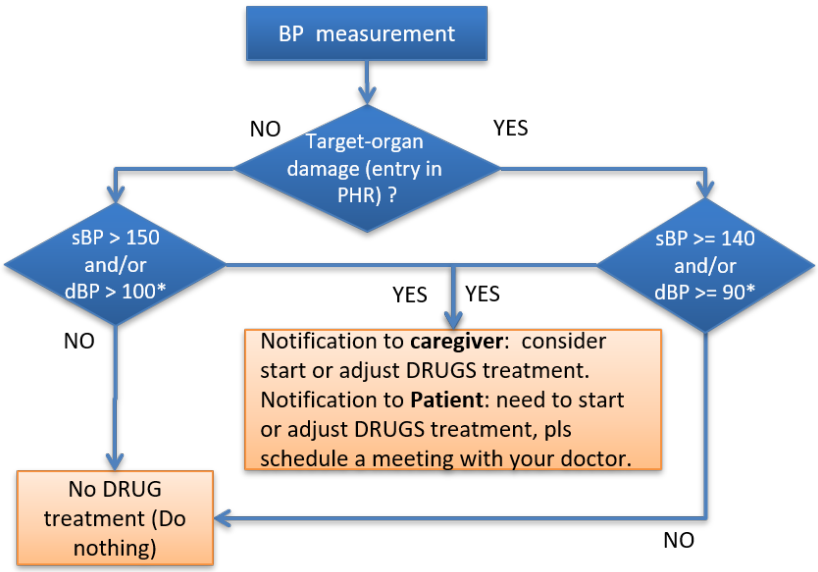

Fig. C.9 Workflow for monitoring blood pressure every 2 days in the context of chronic hypertension (source: [25]).

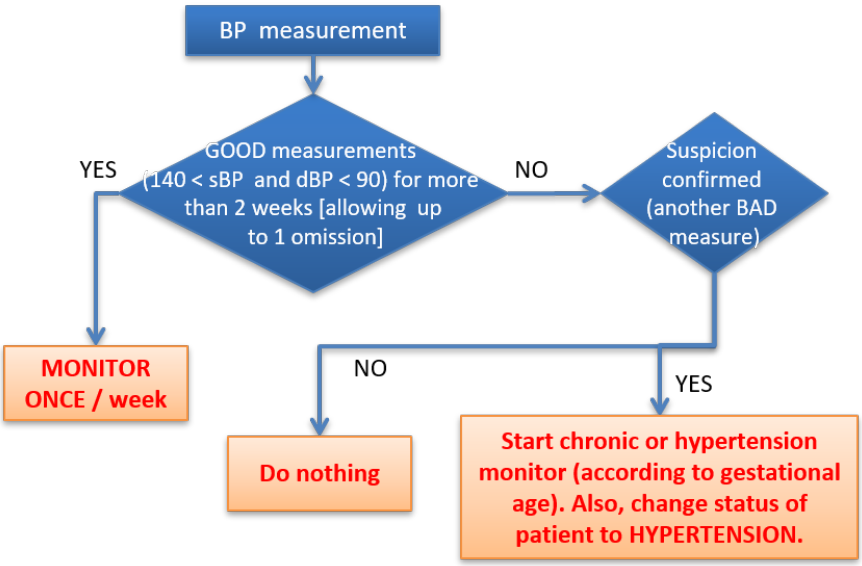

Fig. C.10 Workflow for monitoring blood pressure twice weekly in the context of normal blood pressure (source: [25]).

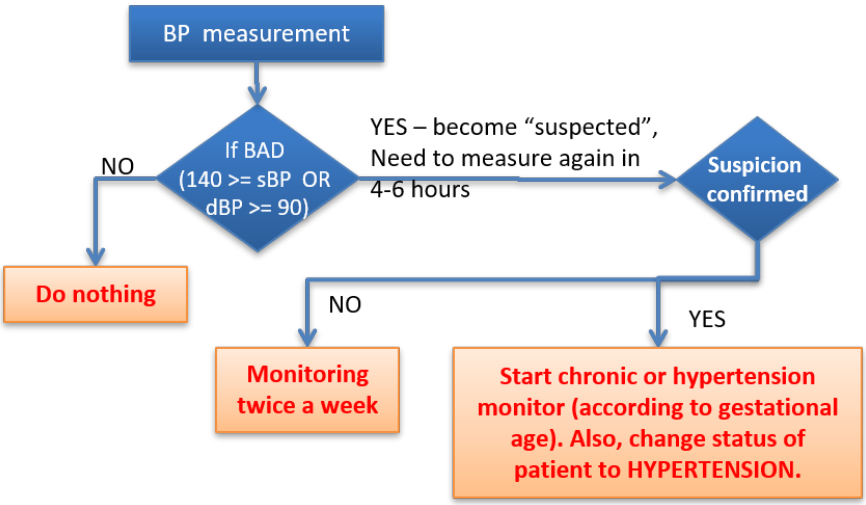

Fig. C.11 Workflow for monitoring blood pressure once weekly in the context of normal blood pressure (source: [25]).

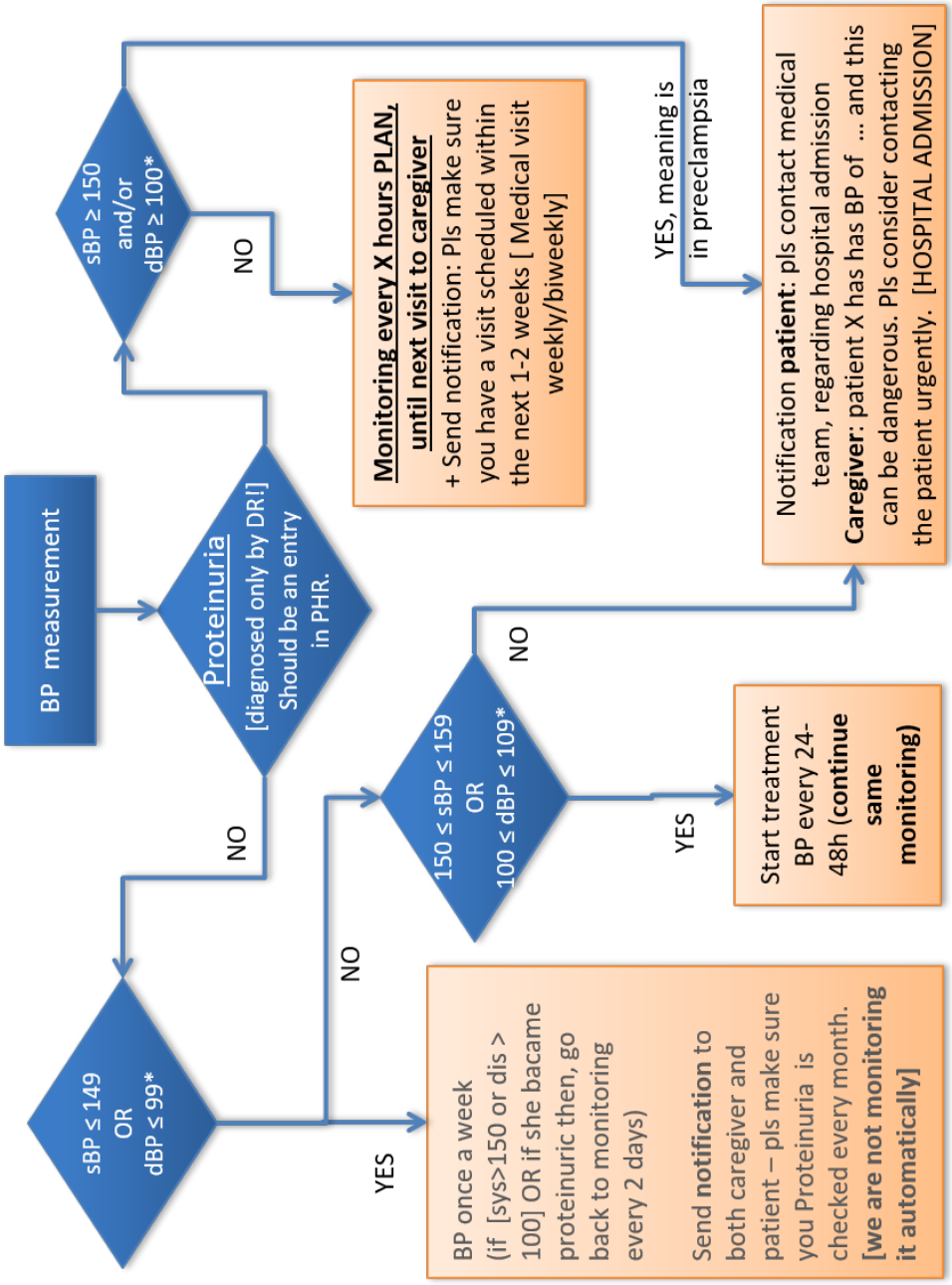

Fig. C.12 Workflow for monitoring blood pressure every two days in the context of gestational hypertension (source: [25]).

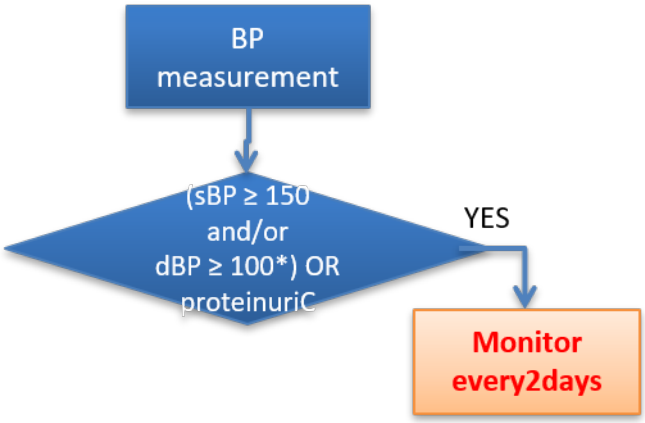

Fig. C.13 Workflow for monitoring blood pressure once weekly in the context of gestational hypertension (source: [25]).

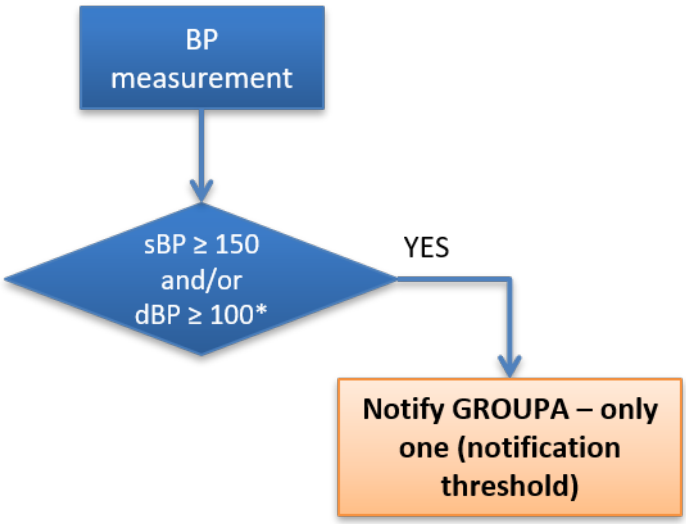

Fig. C.14 Workflow for monitoring blood pressure every few hours in the context of gestational hypertension (source: [25]).

Appendix D

Formalised Guideline for Gestational Diabetes Mellitus

The following is an overview (in the form of comments) of the complete source code that resulted from formalising the GDM guideline (Appendix C) using the MADE archetype and guideline languages. The complete source code, including the comments below, is available at https://github.com/nlsfung/MADE-Language/tree/master/exp/gdm and is divided into two files: `GdmIM.rkt` which contains the 50 archetypes that constitute the domain information model for GDM, and `GdmPM.rkt` which contains the 55 MADE processes that constitute all the processes specified in the GDM guideline.

D.1 GDM Domain Information Model (`GdmIM.rkt`)

```
; No measurement types are specified by the guideline.

; Eight different types of observations can be identified from
; the GDM workflows, namely for capturing:
; 1) Blood glucose levels (blood-glucocse).
; 2) Urinary ketone levels (urinary-ketone) (referred to as
;    ketonuria in the guideline but renamed here avoid
;    confusion with the ketonuria abstraction).
```

```
; 3) Events of meals (meal-event) (which is implicitly required
;    in the guideline to detect abnormal blood glucose
;    measurements).
; 4) Carbohydrates intake (carbohydrates-compliance) of a
;    single meal (which is qualified relative to the recommended
;    intake levels).
; 5) Exercise intensity (exercise-intensity) (in terms of METs).
; 6) Systolic blood pressure (systolic-blood-pressure).
; 7) Diastolic blood pressure (diastolic-blood-pressure) (which
;    must be distinguished from systolic BP as each observation
;    can only be associated with one value).
; 8) Events of conception, i.e. becoming pregnant
;    (conception-event) (which is required to determine the
;    gestational age of the patient).

; Eight different types of abstractions can be identified from
; the GDM workflows, namely for capturing:
; 1) Degree of glycemic control (glycemic-control).
; 2) Severity of ketonuria in the patient (ketonuria).
; 3) Degree of non-compliance to the recommended carbohydrates
;    intake (carbohydrates-compliance).
; 4) Exercise compliance in the resting context
;    (exercise-compliance-resting).
; 5) Exercise compliance in the active context
;    (exercise-compliance-active).
; 6) Degree of hypertension (hypertension).
; 7) Target organ damage (target-organ-damage).
; 8) Proteinuria (proteinuria).

; Three different types of action instructions can be
; identified from the GDM workflows, namely for effectuating:
; 1) Administrations of insulin, which is assumed to be
;    measured in International Units of insulin
```

```
;    (administer-insulin-action).
; 2) Changes in the diet (i.e. nutritional prescription),
;    which is assumed to be measured in grams of carbohydrates
;    (change-diet-action).
; 3) Increases in carbohydrates intake at dinner
;    (change-dinner-action). It is assumed that this instruction
;    can occur in conjunction with instruction 2).

; 17 different types of control instructions can be identified
; from the GDM workflows, namely for controlling:
; 1) Blood glucose (BG) measurements (monitor-bg-control).
; 2) The workflow associated with changing nutritional
;    prescription (bg-nutrition-change-control).
; 3) The workflow associated with starting or changing insulin
;    therapy (bg-insulin-control).
; 4) The workflow associated with monitoring BG for two days
;    each week (bg-twice-weekly-control).
; 5) The workflow associated with monitoring BG every day
;    (bg-daily-control).
; 6) Urinary ketone measurements (monitor-uk-control).
; 7) The workflow associated with increasing dincarbohydrates
;    intake at dinner (uk-dinner-increase-control).
; 8) The workflow associated with monitoring urinary ketones
;    twice weekly (uk-twice-weekly-control).
; 9) The workflow associated with monitoring urinary ketones
;    daily (uk-daily-control).
; 10) Systolic blood pressure measurements
;     (monitor-systolic-bp-control).
; 11) Diastolic blood pressure measurements
;     (monitor-diastolic-bp-control).
; 12) The workflow associated with monitoring blood pressure
;     once each week (bp-once-weekly-control).
; 13) The workflow associated with monitoring blood pressure
```

```
;     twice each week (bp-twice-weekly-control).
; 14) The workflow associated with chronic hypertension
;     (bp-chronic-control).
; 15) The workflow associated with gestational hypertension
;     (every 2 days) (bp-gestational-control).
; 16) The workflow associated with gestational hypertension
;     (every week) (bp-once-weekly-gestational-control).
; 17) The workflow associated with deciding to monitor BP
;     every few hours (bp-hourly-gestational-control).

; The twenty different instruction archetypes constitute
; 14 different types of action plans, namely for:
; 1) Monitoring blood glucose for two days every week
;    (bg-twice-weekly-plan).
; 2) Adjusting the prescribed insulin therapy
;    (adjust-insulin-plan).
; 3) Changing the nutritional prescription
;    (change-nutrition-plan).
; 4) Starting insulin therapy (start-insulin-plan).
; 5) Monitoring blood glucose daily (bg-daily-plan).
; 6) Monitoring urinary ketones twice a week
;    (uk-twice-weekly-plan).
; 7) Changing the carbohydrates intake at dinner (or before
;    bed-time) (increase-dinner-intake-plan)
; 8) Monitoring urinary ketones daily (uk-daily-plan).
; 9) Monitoring blood pressure once a week (for no hypertension)
;    (bp-once-weekly-plan).
; 10) Monitoring blood pressure in the context of chronic
;     hypertension (chronic-hypertension-plan).
; 11) Monitoring blood pressure every two days for gestational
;     hypertension (gestational-hypertension-plan).
; 12) Monitoring blood pressure twice a week (for no
;     hypertension) (bp-twice-weekly-plan).
```

```
; 13) Monitoring blood pressure once a week (for gestational
;     hypertension) (gestational-weekly-plan).
; 14) Monitoring blood pressure every few hours in the context
;     of gestational hypertension (gestational-hours-plan).
```

D.2 GDM Domain Process Model (GdmPM.rkt)

```
; No monitoring processes are specified by the guideline.

; analyse-blood-glucose (ABG) analyses blood glucose (BG)
; measurements to determine the patient's degree of glycemic
; control, which can be:
; a) Good (BG levels are normal for a month)
; b) Poor (A single abnormal BG value)
; c) Meal-compliance poor (>= 2 abnormal BG values in a week but
;    diet compliant).
; d) Meal-incompliance poor (>= 2 abnormal BG values due to diet
;    incompliance).
; e) Non-related poor (>= 2 abnormal BG values at different
;    intervals)
; f) Very poor (>= 2 abnormal BG values exceeding a given
;    threshold).

; analyse-urinary-ketone (AUK) analyses urinary ketone (UK)
; measurements to detect the patient's degree of ketonuria,
; which can be:
; a) Negative (UK levels are negative in 1 week)
; b) Positive (> 2 positive results in 1 week)

; analyse-carbohydrates-intake (ACI) analyses carbohydrates
; intake (CI) of patient to determine his or her degree of
; compliance to the pre-determined diet, which can be:
; a) Insufficient (At least one single negative CI level in a
```

```
;    week)
; b) Non-compliant (At least two non-compliance in a week)

; analyse-blood-pressure (ABP) analyses blood pressure (BP)
; measurements, both systolic and diastolic, to determine the
; patient's degree of high blood glucose, which can be:
; a) High (sBP >= 140 and/or dBP >= 90)
; b) Very high (sBP >= 150 and/or dBP >= 100)
; c) Normal (sBP < 140 and dBP < 90 for more than 2 weeks)
; d) Sustained high (2 high measurements in 6 hours)
; e) Extremely high (sBP >= 160 and dBP >= 110)

; decide-bg-twice-weekly (DBg2Wk) relates to the decision to
; adjust blood glucose monitoring to two days each week instead
; of daily. The decision criteria involves the following
; abstraction(s):
; 1) glycemic-control ('good)
; It affects the following actions and processes:
; 1) monitor-blood-glucose (two days every week)
; 2) decide-bg-nutrition-change (disabled)
; 3) decide-bg-insulin (disabled)
; 4) decide-bg-twice-weekly (disabled)
; 5) decide-bg-daily (enabled after 7 days)

; decide-bg-daily (DBgDaily) relates to the decision to adjust
; blood glucose monitoring to daily (instead of two days each
; week). The decision criteria involves the following
; abstraction(s):
; 1) glycemic-control (2 abnormal values in a week)
; It affects the following actions and processes:
; 1) monitor-blood-glucose (daily)
; 2) decide-bg-nutrition-change (enabled after 7 days)
; 3) decide-bg-insulin (enabled after 7 days)
```

```
; 4) decide-bg-twice-weekly (enabled after 7 days)
; 5) decide-bg-daily (disabled)

; decide-bg-insulin (DBgInsulin) is a proxy process for deciding
; to start insulin therapy. The decision criteria involves the
; following abstraction(s):
; 1) glycemic-control (not 'good or 'poor)
; 2) ketonuria ('positive)
; It affects the following actions and processes:
; 1) decide-bg-nutrition-change (disabled)
; 2) decide-bg-twice-weekly (disabled)
; 3) decide-bg-insulin (disabled)
; 4) decide-bg-twice-weekly-post-insulin (enabled after 7 days)
; 5) decide-bg-insulin-adjust (enabled after 7 days)
; 6) administer-insulin-action (4 times each day)
; (Note: Since the guideline does not specify the amount of
; insulin to prescribe, its set to an arbitrary value of -1.

; decide-bg-twice-weekly-post-insulin (DBg2WkPostInsulin)
; relates to the decision to adjust blood glucose monitoring to
; two days each week instead of daily (after the prescription of
; insulin). The decision criteria involves the following
; abstraction(s):
; 1) glycemic-control ('good)
; It affects the following actions and processes:
; 1) monitor-blood-glucose (two days every week)
; 2) decide-bg-insulin-adjust (disabled)
; 3) decide-bg-twice-weekly-post-insulin (disabled)
; 4) decide-bg-daily-post-insulin (enabled after 7 days)

; decide-bg-daily-post-insulin (DBgDailyPostInsulin) relates to
; the decision to adjust blood glucose monitoring to daily
; (instead of two days each week) after the prescription of
```

```
; insulin. The decision criteria involves the following
; abstraction(s):
; 1) glycemic-control (2 abnormal values in a week)
; It affects the following actions and processes:
; 1) monitor-blood-glucose (daily)
; 2) decide-bg-insulin-adjust (enabled after 7 days)
; 3) decide-bg-twice-weekly-post-insulin (enabled after 7 days)
; 4) decide-bg-daily-post-insulin (disabled)

; decide-bg-insulin-adjust (DBgInsAdjust) is a proxy process for
; adjusting the insulin therapy for the patient. The decision
; criteria involves the following abstraction(s):
; 1) glycemic-control (1 abnormal value detected).
; It affects the following actions and processes:
; 1) administer-insulin-action.

; decide-bg-nutrition-change (DBgCarb) is a proxy process for
; changing the nutritional prescription of the patient due to
; poor glycemic control. Specifically, the decision criteria
; involves the following abstraction(s):
; 1) glyemic-control ('meal-compliant-poor)
; 2) ketonuria ('negative)
; It affects the following actions and processes:
; 1) decide-bg-twice-weekly (disabled)
; 2) decide-bg-nutrition-change (disabled)
; 3) decide-bg-insulin (disabled)
; 4) decide-bg-twice-weekly-post-nutrition (enabled after 7 days)
; 5) decide-bg-insulin-post-nutrition (enabled after 7 days)
; 6) change-diet-action (4 times each day)
; Note: Since the guideline does not specify a concrete
; nutrition change, its set to an arbitrary value of -1.

; decide-bg-twice-weekly-post-nutrition (DBg2WkPostNutrition)
```

```
; relates to the decision to adjust blood glucose monitoring to
; two days each week instead of daily (after the changing
; nutrition prescription). The decision criteria involves the
; following abstraction(s):
; 1) glycemic-control ('good)
; It affects the following actions and processes:
; 1) monitor-blood-glucose (two days every week)
; 2) decide-bg-twice-weekly-post-nutrition (disabled)
; 3) decide-bg-insulin-post-nutrition (disabled)
; 4) decide-bg-daily-post-nutrition (enabled after 7 days)

; decide-bg-daily-post-nutrition (DBgDailyPostNutrition) relates
; to the decision to adjust blood glucose monitoring to daily
; (instead of two days each week) after the changing nutrition
; prescription. The decision criteria involves the following
; abstraction(s):
; 1) glycemic-control (2 abnormal values in a week)
; It affects the following actions and processes:
; 1) monitor-blood-glucose (daily)
; 2) decide-bg-insulin-post-nutrition (enabled after 7 days)
; 3) decide-bg-twice-weekly-post-nutrition (enabled after 7 days)
; 4) decide-bg-daily-post-nutrition (disabled)

; decide-bg-insulin-post-nutrition (DBgInsulinPostNutrition) is
; a proxy process for deciding to start insulin therapy (after
; changing nutrition prescription).
; The decision criteria involves the following abstraction(s):
; 1) glycemic-control (not 'good or 'poor)
; It affects the following actions and processes:
; 1) decide-bg-twice-weekly-post-nutrition (disabled)
; 2) decide-bg-insulin-post-nutrition (disabled)
; 3) decide-bg-twice-weekly-post-insulin (enabled after 7 days)
; 4) decide-bg-insulin-adjust (enabled after 7 days)
```

```
; 5) administer-insulin-action (4 times each day)
; (Note: Since the guideline does not specify the amount of
; insulin to prescribe, its set to an arbitrary value of -1.

; decide-uk-twice-weekly (DUk2Wk) relates to the decision to
; adjust urinary ketone monitoring to two days each week instead
; of daily. The decision criteria involves the following
; abstraction(s):
; 1) ketonuria ('negative)
; It affects the following actions and processes:
; 1) monitor-urinary-ketones (two days every week)
; 2) decide-uk-dinner-increase (disabled)
; 3) decide-uk-twice-weekly (disabled)
; 4) decide-uk-daily (enabled after 7 days)

; decide-uk-daily (DUkDaily) relates to the decision to adjust
; urinary ketone monitoring to daily (instead of two days each
; week). The decision criteria involves the following
; abstraction(s):
; 1) ketonuria ('positive)
; It affects the following actions and processes:
; 1) monitor-urinary-ketones (daily)
; 2) decide-uk-dinner-increase (enabled after 7 days)
; 3) decide-uk-twice-weekly (enabled after 7 days)
; 4) decide-uk-daily (disabled)

; decide-uk-dinner-increase (DUkCarb) relates to the decision to
; increase carbohydrates intake at dinner. The decision criteria
; involves the following abstraction(s):
; 1) ketonuria ('positive)
; 2) carbohydrates-compliance (not 'insufficient)
; It affects the following actions and processes:
; 1) decide-uk-dinner-increase (disabled)
```

```
; 2) decide-uk-twice-weekly (disabled)
; 3) decide-uk-daily (disabled)
; 4) decide-uk-twice-weekly-post-dinner (enabled after 7 days)
; 5) change-dinner-action (daily)

; decide-uk-twice-weekly-post-dinner (DUk2WkPostDinner) relates
; to the decision to adjust urinary ketone monitoring to two
; days each week instead of daily (after increasing carbohydrates
; intake at dinner). The decision criteria involves the following
; abstraction(s):
; 1) ketonuria ('negative)
; It affects the following actions and processes:
; 1) monitor-urinary-ketones (two days every week)
; 3) decide-uk-twice-weekly-post-dinner (disabled)
; 4) decide-uk-daily-post-dinner (enabled after 7 days)

; decide-uk-daily-post-dinner (DUkDailyPostDinner) relates to
; the decision to adjust urinary ketone monitoring to daily
; (instead of two days each week) after increasing carbohydrates
; intake. The decision criteria involves the following
; abstraction(s):
; 1) ketonuria ('positive)
; It affects the following actions and processes:
; 1) monitor-urinary-ketones (daily)
; 2) decide-uk-twice-weekly-post-dinner (enabled after 7 days)
; 3) decide-uk-daily-post-dinner (disabled)

; decide-bp-once-weekly (DBp1Wk) relates to the decision to
; adjust blood pressure monitoring to once a week instead of
; twice a week. The decision criteria involves the following
; abstraction(s):
; 1) hypertension ('normal)
; It affects the following actions and processes:
```

```
; 1) monitor-blood-pressure (once every week)
; 2) decide-bp-once-weekly (disabled)
; 3) decide-bp-twice-weekly (enabled after 7 days)

; decide-bp-twice-weekly (DBp2Wk) relates to the decision to
; adjust blood pressure monitoring to twice a week instead of
; once a week. The decision criteria involves the following
; abstraction(s):
; 1) hypertension ('high)
; It affects the following actions and processes:
; 1) monitor-blood-pressure (twice every week)
; 2) decide-bp-once-weekly (enabled after 7 days)
; 3) decide-bp-twice-weekly (disabled)

; decide-bp-chronic (DBpChronic) is a proxy process for starting
; the chronic blood pressure monitoring plan. The decision
; criteria involves the following abstraction(s):
; 1) hypertension ('sustained-high)
; It affects the following actions and processes:
; 1) monitor-blood-pressure (every two days)
; 2) decide-bp-once-weekly (disabled)
; 3) decide-bp-twice-weekly (disabled)
; 4) decide-bp-chronic (disabled)

; decide-bp-gestational (DBpGestational) is a proxy process for
; starting the gestational blood pressure monitoring plan. The
; decision criteria involves the following abstraction(s):
; 1) hypertension ('sustained-high)
; It affects the following actions and processes:
; 1) monitor-blood-pressure (every two days)
; 2) decide-bp-once-weekly (disabled)
; 3) decide-bp-twice-weekly (disabled)
; 4) decide-bp-gestational (disabled)
```

```
; 5) decide-bp-once-weekly-gestational (enabled after 7 days)
; 6) decide-bp-hourly-gestational (enabled after 7 days)

; decide-bp-once-weekly-gestational (DBp1WkGestational) relates
; to the decision to adjust blood pressure monitoring to once a
; week instead of every two days (in the gestational hypertension
; plan). The decision criteria involves the following
; abstraction(s):
; 1) hypertension (not 'very-high or 'extremely-high)
; 2) proteinuria (false)
; It affects the following actions and processes:
; 1) monitor-blood-pressure (once a week)
; 2) decide-bp-once-weekly-gestational (disabled)
; 3) decide-bp-hourly-gestational (disabled)
; 4) decide-bp-two-days-gestational (enabled after 7 days)

; decide-bp-two-days-gestational (DBp2DaysGest) relates to
; decision to adjust blood pressure monitoring to every two days
; instead of once a week (in the gestational hypertension
; workflow). It is equivalent to the process decide-
; bp-gestational except that the decision criteria involves:
; 1) hypertension ('very-high or 'extremely-high)
; 2) proteinuria (true)
; It affects the following actions and processes:
; 1) monitor-blood-pressure (every two days)
; 2) decide-bp-once-weekly-gestational (enabled after 7 days)
; 3) decide-bp-hourly-gestational (enabled after 7 days)
; 4) decide-bp-two-days-gestational (disabled)

; decide-bp-hourly-gestational (DBpHoursGest) is a proxy process
; for deciding to adjust blood pressure monitoring to every few
; hours (e.g. 4) instead of once a week (in the gestational
; hypertension workflow). The decision criteria involves:
```

```
; 1) hypertension (not 'very-high or 'extremely-high)
; 2) proteinuria (true)
; It affects the following actions and processes:
; 1) monitor-blood-pressure (every 4 hours)
; 2) decide-bp-once-weekly-gestational (disabled)
; 3) decide-bp-hourly-gestational (disabled)

; effectuate-administer-insulin is a proxy process for
; administering insulin.

; effectuate-change-diet is responsible for effectuating the
; change to the patient's diet (due to poor glycaemic control).

; effectuate-change-dinner is responsible for effectuating the
; change to the patient's carbohydrates intake at dinner (due to
; positive ketonuria).

; effectuate-monitor-bg is responsible for effectuating any
; changes to the monitoring of blood glucose.

; effectuate-bg-nutrition-change is responsible for effectuating
; the decide-bg-nutrition-change process.

; effectuate-bg-insulin-control and its two variants,
; effectuate-bg-insulin-post-nutrition-control and
; effectuate-bg-insulin-adjust-control, are responsible for
; effectuating the 'decide-bg-insulin process.

; effectuate-bg-twice-weekly-control and its two variants,
; effectuate-bg-twice-weekly-post-nutrition-control and
; effectuate-bg-twice-weekly-post-insulin-control, are responsible
; for effectuating the decide-bg-twice-weekly process.
```

```
; effectuate-bg-daily-control and its two variants,
; effectuate-bg-daily-post-nutrition-control and
; effectuate-bg-daily-post-insulin-control, are responsible for
; effectuating the decide-bg-daily process.

; effectuate-monitor-uk-control is responsible for effectuating
; changes to the monitoring of urinary ketones (uk).

; effectuate-uk-dinner-increase is responsible for effectuating
; the 'decide-uk-dinner-increase process.

; effectuate-uk-twice-weekly-control and its variant,
; effectuate-uk-twice-weekly-post-dinner-control, are responsible
; for effectuating the 'decide-uk-twice-weekly process.

; effectuate-uk-daily-control and its variant,
; effectuate-uk-daily-post-dinner-control, are responsible for
; effectuating the 'decide-uk-daily process.

; effectuate-monitor-systolic-bp-control is responsible for
; effectuating changes to the monitoring of systolic blood
; pressure (bp).

; effectuate-monitor-diastolic-bp-control is responsible for
; effectuating changes to the monitoring of diastolic blood
; pressure (bp).

; effectuate-bp-once-weekly-control is responsible for
; effectuating the decide-bp-once-weekly process.

; effectuate-bp-twice-weekly-control is responsible for
; effectuating the decide-bp-twice-weekly process.
```

```
; effectuate-bp-chronic-control is responsible for effectuating
; the decide-bp-chronic process.

; effectuate-bp-gestational-control and its variant,
; effectuate-bp-two-days-gestational-control, are responsible for
; effectuating the decide-bp-gestational process.

; effectuate-bp-once-weekly-gestational-control is responsible
; for effectuating the 'decide-bp-once-weekly-gestational
; process.

; effectuate-bp-hourly-gestational-control is responsible for
; effectuating the 'decide-bp-hourly-gestational process.
```

Appendix E

Verification of the Formalised Guideline

The following is the source code used to verify the MADE processes that resulted from formalising the GDM guideline. This source code is also included as comments in https://github.com/nlsfung/MADE-Language/tree/master/exp/gdm/GdmPM.rkt, and it details the data used to verify each MADE process. Note that for variants of the same process, the same inputs were used to verify them. Thus for simplicity, they are not included within this appendix.

```
(verify-process
   analyse-blood-glucose
   (list (generate-list
          blood-glucose
          (datetime 2019 12 2 7 0 0)
          (datetime 2019 12 3 24 0 0)
          (duration 0 12 0 0))
         (generate-list
          meal-event
          (datetime 2019 12 2 6 0 0)
          (datetime 2019 12 3 24 0 0)
          (duration 0 12 0 0))
```

```
         (generate-list
          carbohydrate-intake
          (datetime 2019 12 2 6 0 0)
          (datetime 2019 12 3 24 0 0)
          (duration 0 12 0 0)))
   (get-datetime (datetime 2019 12 3 19 0 0)
                 (datetime 2019 12 3 19 0 0)))

(verify-process
   analyse-urinary-ketone
   (list (generate-list
          urinary-ketone
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 31 24 0 0)
          5))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 31 24 0 0)))

(verify-process
   analyse-carbohydrates-intake
   (list (generate-list
          carbohydrate-intake
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 31 24 0 0)
          5))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 31 24 0 0)))

(verify-process
   analyse-blood-pressure
   (list (generate-list
          systolic-blood-pressure
          (datetime 2019 12 1 0 0 0)
```

```
46          (datetime 2019 12 15 24 0 0)
47          2)
48         (generate-list
49          diastolic-blood-pressure
50          (datetime 2019 12 1 0 0 0)
51          (datetime 2019 12 15 24 0 0)
52          2))
53    (get-datetime (datetime 2019 12 1 0 0 0)
54                  (datetime 2019 12 15 24 0 0)))
55
56 (verify-process
57    decide-bg-twice-weekly
58    (list (generate-list
59          glycemic-control
60          (datetime 2019 12 1 0 0 0)
61          (datetime 2019 12 15 24 0 0)
62          2))
63    (get-datetime (datetime 2019 12 1 0 0 0)
64                  (datetime 2019 12 15 24 0 0)))
65
66 (verify-process
67    decide-bg-daily
68    (list (generate-list
69          glycemic-control
70          (datetime 2019 12 1 0 0 0)
71          (datetime 2019 12 15 24 0 0)
72          2))
73    (get-datetime (datetime 2019 12 1 0 0 0)
74                  (datetime 2019 12 15 24 0 0)))
75
76 (verify-process
77    decide-bg-insulin
78    (list (generate-list
```

```
          glycemic-control
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2)
         (generate-list
          ketonuria
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bg-twice-weekly-post-insulin
   (list (generate-list
          glycemic-control
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bg-daily-post-insulin
   (list (generate-list
          glycemic-control
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
```

```
112     decide-bg-insulin-adjust
113     (list (generate-list
114            glycemic-control
115            (datetime 2019 12 1 0 0 0)
116            (datetime 2019 12 15 24 0 0)
117            2))
118     (get-datetime (datetime 2019 12 1 0 0 0)
119                   (datetime 2019 12 15 24 0 0)))
120
121 (verify-process
122     decide-bg-nutrition-change
123     (list (generate-list
124            glycemic-control
125            (datetime 2019 12 1 0 0 0)
126            (datetime 2019 12 15 24 0 0)
127            2)
128           (generate-list
129            ketonuria
130            (datetime 2019 12 1 0 0 0)
131            (datetime 2019 12 15 24 0 0)
132            2))
133     (get-datetime (datetime 2019 12 1 0 0 0)
134                   (datetime 2019 12 15 24 0 0)))
135
136 (verify-process
137     decide-bg-twice-weekly-post-nutrition
138     (list (generate-list
139            glycemic-control
140            (datetime 2019 12 1 0 0 0)
141            (datetime 2019 12 15 24 0 0)
142            2))
143     (get-datetime (datetime 2019 12 1 0 0 0)
144                   (datetime 2019 12 15 24 0 0)))
```

```
(verify-process
   decide-bg-daily-post-nutrition
   (list (generate-list
          glycemic-control
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bg-insulin-post-nutrition
   (list (generate-list
          glycemic-control
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-uk-twice-weekly
   (list (generate-list
          ketonuria
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-uk-daily
```

```
178     (list (generate-list
179            ketonuria
180            (datetime 2019 12 1 0 0 0)
181            (datetime 2019 12 15 24 0 0)
182            2))
183     (get-datetime (datetime 2019 12 1 0 0 0)
184                   (datetime 2019 12 15 24 0 0)))
185
186 (verify-process
187     decide-uk-dinner-increase
188     (list (generate-list
189            ketonuria
190            (datetime 2019 12 1 0 0 0)
191            (datetime 2019 12 15 24 0 0)
192            2)
193           (generate-list
194            carbohydrates-compliance
195            (datetime 2019 12 1 0 0 0)
196            (datetime 2019 12 15 24 0 0)
197            2))
198     (get-datetime (datetime 2019 12 1 0 0 0)
199                   (datetime 2019 12 15 24 0 0)))
200
201 (verify-process
202     decide-uk-twice-weekly-post-dinner
203     (list (generate-list
204            ketonuria
205            (datetime 2019 12 1 0 0 0)
206            (datetime 2019 12 15 24 0 0)
207            2))
208     (get-datetime (datetime 2019 12 1 0 0 0)
209                   (datetime 2019 12 15 24 0 0)))
210
```

```
(verify-process
   decide-uk-daily-post-dinner
   (list (generate-list
          ketonuria
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bp-once-weekly
   (list (generate-list
          hypertension
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bp-twice-weekly
   (list (generate-list
          hypertension
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bp-chronic
   (list (generate-list
```

```
          hypertension
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bp-gestational
   (list (generate-list
          hypertension
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bp-once-weekly-gestational
   (list (generate-list
          hypertension
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2)
         (generate-list
          proteinuria
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
```

```
   decide-bp-two-days-gestational
   (list (generate-list
          hypertension
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2)
         (generate-list
          proteinuria
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
   decide-bp-hourly-gestational
   (list (generate-list
          hypertension
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2)
         (generate-list
          proteinuria
          (datetime 2019 12 1 0 0 0)
          (datetime 2019 12 15 24 0 0)
          2))
   (get-datetime (datetime 2019 12 1 0 0 0)
                 (datetime 2019 12 15 24 0 0)))

(verify-process
 effectuate-administer-insulin
 (list (generate-list
        adjust-insulin-plan
```

```
310         (datetime 2019 12 7 0 0 0)
311         (datetime 2019 12 7 24 0 0)
312         1))
313  (get-datetime (datetime 2019 12 7 20 0 0)
314                (datetime 2019 12 7 20 0 0)))
315
316 (verify-process
317  effectuate-administer-insulin
318  (list (generate-list
319         start-insulin-plan
320         (datetime 2019 12 7 0 0 0)
321         (datetime 2019 12 7 24 0 0)
322         1))
323  (get-datetime (datetime 2019 12 7 20 0 0)
324                (datetime 2019 12 7 20 0 0)))
325
326 (verify-process
327    effectuate-change-diet
328    (list (generate-list
329           change-nutrition-plan
330           (datetime 2019 12 7 0 0 0)
331           (datetime 2019 12 7 24 0 0)
332           1))
333    (get-datetime (datetime 2019 12 7 20 0 0)
334                  (datetime 2019 12 7 20 0 0)))
335
336 (verify-process
337    effectuate-change-dinner
338    (list (generate-list
339           increase-dinner-intake-plan
340           (datetime 2019 12 7 0 0 0)
341           (datetime 2019 12 7 24 0 0)
342           1))
```

```
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-monitor-bg
   (list (generate-list
          bg-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          bg-daily-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bg-nutrition-change
   (list (generate-list
          bg-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          change-nutrition-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          start-insulin-plan
          (datetime 2019 12 7 0 0 0)
```

```
376            (datetime 2019 12 7 24 0 0)
377            1)
378           (generate-list
379            bg-daily-plan
380            (datetime 2019 12 7 0 0 0)
381            (datetime 2019 12 7 24 0 0)
382            1))
383     (get-datetime (datetime 2019 12 7 20 0 0)
384                   (datetime 2019 12 7 20 0 0)))
385
386 (verify-process
387     effectuate-bg-insulin-control
388     (list (generate-list
389            bg-twice-weekly-plan
390            (datetime 2019 12 7 0 0 0)
391            (datetime 2019 12 7 24 0 0)
392            1)
393           (generate-list
394            change-nutrition-plan
395            (datetime 2019 12 7 0 0 0)
396            (datetime 2019 12 7 24 0 0)
397            1)
398           (generate-list
399            start-insulin-plan
400            (datetime 2019 12 7 0 0 0)
401            (datetime 2019 12 7 24 0 0)
402            1)
403           (generate-list
404            bg-daily-plan
405            (datetime 2019 12 7 0 0 0)
406            (datetime 2019 12 7 24 0 0)
407            1))
408     (get-datetime (datetime 2019 12 7 20 0 0)
```

```
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bg-twice-weekly-control
   (list (generate-list
          bg-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          change-nutrition-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          start-insulin-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          bg-daily-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bg-daily-control
   (list (generate-list
          bg-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
```

```
          1)
         (generate-list
          bg-daily-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-monitor-uk-control
   (list (generate-list
          uk-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          uk-daily-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-uk-dinner-increase
   (list (generate-list
          uk-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          uk-daily-plan
```

```
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          increase-dinner-intake-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-uk-twice-weekly-control
   (list (generate-list
          uk-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          uk-daily-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          increase-dinner-intake-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-uk-daily-control
```

```
   (list (generate-list
          uk-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          uk-daily-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          increase-dinner-intake-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-monitor-systolic-bp-control
   (list (generate-list
          bp-once-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          bp-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          chronic-hypertension-plan
          (datetime 2019 12 7 0 0 0)
```

```
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-hours-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-monitor-diastolic-bp-control
   (list (generate-list
          bp-once-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          bp-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
```

```
          chronic-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-hours-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bp-once-weekly-control
   (list (generate-list
          bp-once-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          bp-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
```

```
          1)
         (generate-list
          chronic-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bp-twice-weekly-control
   (list (generate-list
          bp-once-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          bp-twice-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          chronic-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-hypertension-plan
```

```
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bp-chronic-control
   (list (generate-list
          chronic-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bp-gestational-control
   (list (generate-list
          gestational-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bp-once-weekly-gestational-control
```

```
   (list (generate-list
          gestational-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-hours-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))

(verify-process
   effectuate-bp-hourly-gestational-control
   (list (generate-list
          gestational-hypertension-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-weekly-plan
          (datetime 2019 12 7 0 0 0)
          (datetime 2019 12 7 24 0 0)
          1)
         (generate-list
          gestational-hours-plan
          (datetime 2019 12 7 0 0 0)
```

```
          (datetime 2019 12 7 24 0 0)
          1))
   (get-datetime (datetime 2019 12 7 20 0 0)
                 (datetime 2019 12 7 20 0 0)))
```