A Relationship-Based Approach to Model Integration

Marsha Chechik[†]   Shiva Nejati[‡]   Mehrdad Sabetzadeh[‡]

[†]Department of Computer Science
University of Toronto, Toronto, ON, Canada
chechik@cs.toronto.edu

[‡]Simula Research Laboratory
Oslo, Norway
{shiva,mehrdad}@simula.no

Abstract

A key problem in model-based development is integrating a collection of models into a single, larger, specification as a way to construct a functional system, to develop a unified understanding, or to enable automated reasoning about properties of the resulting system.

In this article, we suggest that the choice of a particular model integration operator depends on the inter-model relationships that hold between individual models. Based on this observation, we distinguish three key integration operators studied in the literature – merge, composition *and* weaving *– and describe each operator along with the notion of relationship that underlies it.*

We then focus on the merge activity and provide a detailed look at the factors that one must consider in defining a merge operator, particularly the way in which the relationships should be captured during merge. We illustrate these factors using two merge operators that we have developed in our earlier work for combining models that originate from distributed teams.

Keywords: Model-Based Development, Model Integration, Merge, Composition, Weaving.

1   Introduction

During any large-scale development, engineers inevitably have to deal with large collections of models representing different perspectives, different versions across time, different variants in a product family, different system components, different development concerns, and so on. A key problem is then to integrate these models into a single, larger, specification that can be used to operationalize the system under development, to arrive at a unified understanding about the system, or to enable automated end-to-end reasoning about properties that the system must satisfy.

For example, consider a simple Hospital Information System (HIS), designed to manage the clinical and administrative functions of a hospital. An HIS has to meet the needs of multiple stakeholders. These include the medical staff, technicians, administrators, and so on. These people have different areas of concern, and model the system from different perspectives. Figure 1 shows a few possible perspectives in an HIS: those of a doctor, two nurses, and a computer system administrator.

The doctor contributes two models, a UML class diagram (M1) and a UML sequence diagram (M2). M1 captures the concepts and associations relevant to the doctor, and M2 describes a scenario concerning the doctor's regular visit to the patients and updating the patients' medical records. Nurse I contributes three models: M3 is a class diagram expressing the concepts and associations in the nurse's domain; M4 is a sequence diagram capturing the patients' daily check in a hospital ward where a nurse evaluates each patient individually and notes her observations in the patient's medical chart; and M5 captures the different states a nurse could be in (resting, ready, in patient's room) and the possible transitions between them. Figure 1 depicts a second state machine model, M6, attributed to Nurse II, whose view on how a nurse can tend to a patient is finer-grained. Finally, the computer systems administrator, who is in charge of ensuring the integrity and performance of the HIS, expresses a requirement that any update made to the persistent data of the system should be logged in a file, so that problems can be tracked in case of a system failure. This requirement is modelled as a rewrite rule for sequence diagrams, shown in model M7. If the left hand side of the rule (denoted L) matches a fragment of a sequence diagram, then that fragment is rewritten with the right hand side of the rule (denoted R).

The models built for a proposed system such as the HIS described above are not stand-alone objects but instead are inter-related and dependent on one another. Some possible relationships include:
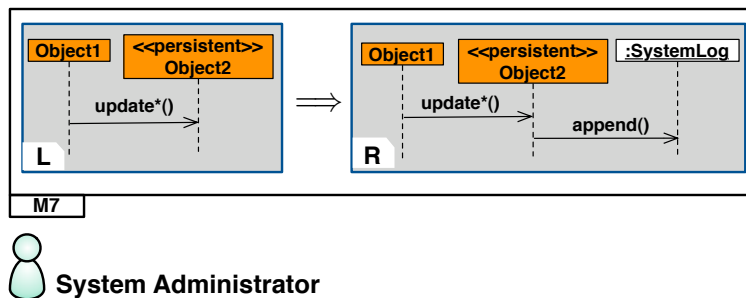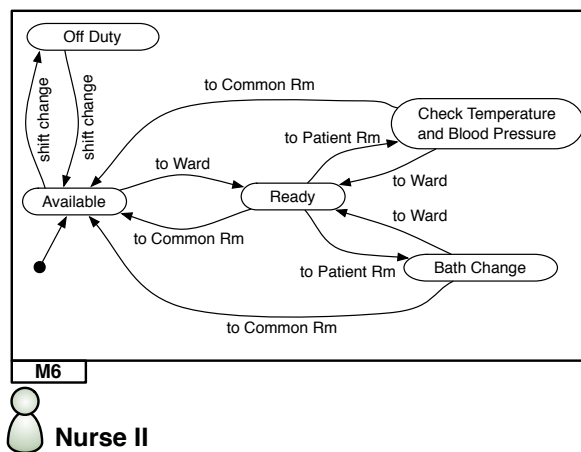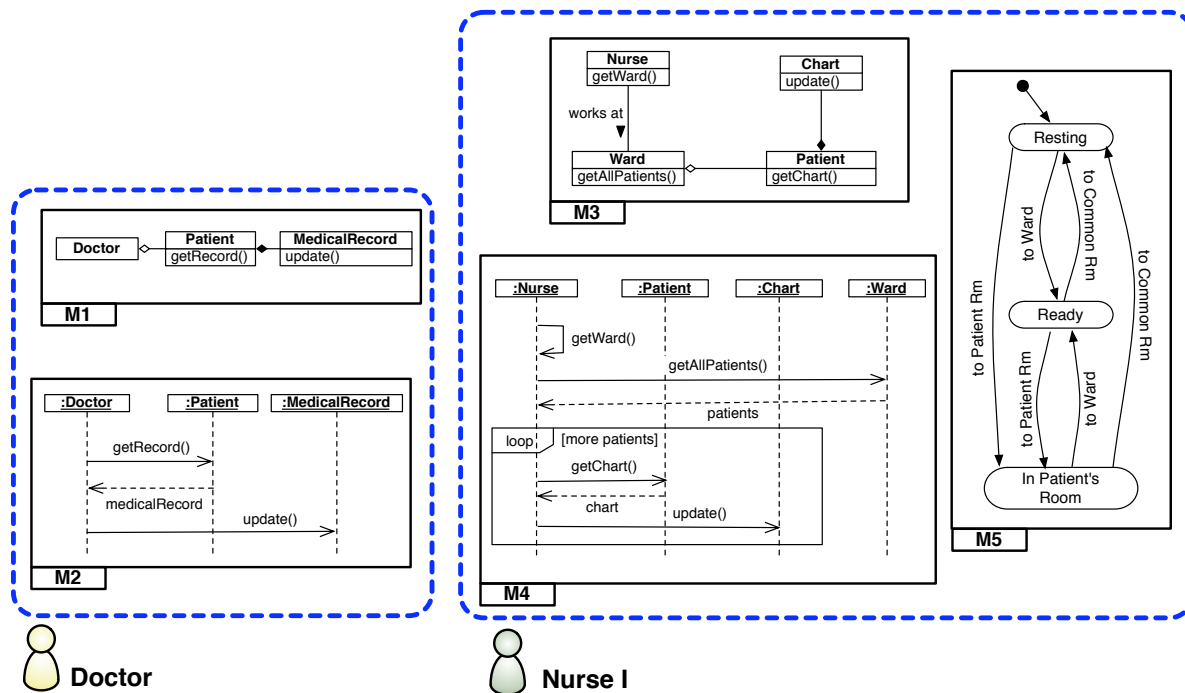
M1

Doctor — Patient
getRecord()
MedicalRecord
update()

M2

:Doctor — :Patient — :MedicalRecord
getRecord()
medicalRecord
update()

M3

Nurse
getWard()
Chart
update()
works at
Ward
getAllPatients()
Patient
getChart()

M4

:Nurse — :Patient — :Chart — :Ward
getWard()
getAllPatients()
patients
loop [more patients]
getChart()
chart
update()

M5

Resting
to Ward
to Common Rm
to Patient Rm
Ready
to Common Rm
to Patient Rm
to Ward
In Patient's Room

Doctor

Nurse I

M6

Off Duty
shift change
shift change
to Common Rm
Check Temperature and Blood Pressure
to Patient Rm
to Ward
Available
to Ward
Ready
to Ward
to Common Rm
Bath Change
to Patient Rm
to Common Rm

Nurse II

M7

L
Object1 — <<persistent>> Object2
update*()

R
Object1 — <<persistent>> Object2 — :SystemLog
update*()
append()

System Administrator

Figure 1. Example models originating from different sources.

2

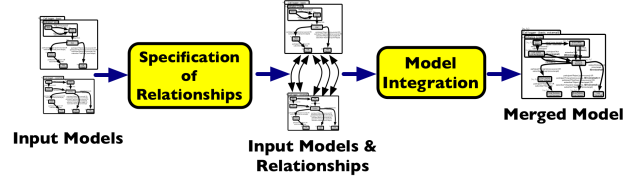**Table 1. Model integration.**

Relationships	Integration operator
Interact	Compose
Cross-cut	Weave
Overlap	Merge

Figure 2. Model integration process.

- the models may *overlap* with one another as information about same or closely associated concepts may appear in multiple models. For example, the Patient class in M1 is likely to be the same as in M3; and Medical-Record in M1 is likely to be the same as Chart in M3. In the same vein, M5 and M6 have overlaps, but Nurse II distinguishes between medical and room service, and differs from Nurse I on whether a nurse is responsible for tending to patients while at rest.

- the models may *interact* (communicate) at run-time. For example, it is possible for a doctor and a nurse to both want to update an individual patient's chart simultaneously. Hence, M2 and M4 may be interacting, in this case, to synchronize accesses to a patient's chart.

- the models may be *cross-cutting*, with some describing ways to alter the behavior or structure of others, in response to different overarching concerns such as safety, security, and performance. For example, M7 cross-cuts M2 and M4, as both models involve updates to persistent objects (MedicalRecord and Chart), hence altering the behaviour of M2 and M4.

Despite the lack of a well-defined terminology, the literature broadly distinguishes three core types of model integration activities, called *merge* [35, 28, 43, 41, 8, 3, 7], *composition* [5, 13, 17, 25] and *weaving* [26, 40, 12, 11]. Given a particular integration context, which of the activities is appropriate? We suggest that the choice of integration activity is determined by the relationship which holds between the models – see Table 1. Merge is used to build a global view of a set of *overlapping* models that capture different angles on the same functionality. Composition is used to assemble a set of autonomous but *interacting* features that run sequentially or in parallel. *Weaving* is used in aspect-oriented development to incorporate *cross-cutting* concerns into a base system.

To emphasize making the complex relationships between the models explicit, we propose an integration process in which relationships are treated as first-class artifacts. The process is shown in Figure 2: Given a set of models, we begin with specifying the relationships between the models. Then, using Table 1, we choose an appropriate integration operator based on the type of the relationships. This produces an integrated model that can be used for further

analysis and development. In Section 2, we instantiate this process to composition, weaving and merge.

In our HIS example, we may decide that we need to (1) merge M1 and M3 to build a complete view on the concepts and relationships in the system; (2) merge M5 and M6 to explicate the commonalities and variabilities between the models; (3) compose M2 and M4 in parallel, so that we can check that composition preserves mutual exclusion to the patient's chart; and (4) weave M7 into M2 and M4 to insert logging behavior for updating persistent objects.

Yet, just explicating the type of relationships may be insufficient to property apply the integration operator using the process in Figure 2. Operationalizing this process for each integration operator requires a close investigation of several other factors, including assumptions about the nature of models, the specific details of the relationships between models, and also the usage and intention of the integration process. In the remainder of the article, we look at the factors that one must consider in defining a merge operator, and specifically, at identifying and capturing overlap relationships (see Section 3). We describe two specific instances of the merge operator that we have developed in our earlier work [35, 28] in response to different contextual needs. We use these operators to provide concrete examples of the influence of the different considerations on the definition of the merge operator (see Section 4). In Section 5, we describe the tool that we have developed in support of these two operators and discuss our experience applying it to combining and reasoning about models coming from distributed teams. We conclude the article in Section 6 with a summary and avenues for future work.

2 Model Integration Operators

In Sections 2.1–2.3, we provide an overview of composition, weaving and merge, respectively, alongside the notion of relationship underlying each of these activities. We use the HIS models in Figure 1 for illustration. In the remainder of the article, M1-M7 refer to the models in that figure. Our goal here is to provide a general understanding of these three different integration operators, without elaborating their details. In Section 2.4, we present a number of important factors that one needs to carefully consider and elaborate when defining an integration operator. We provide

3

a detailed discussion of these factors for the merge activity in Section 3.

2.1 Compose

Composition refers to the process of assembling a set of autonomous but interacting models that capture different components of a system. While the term "composition" is overloaded, in this article we use it to represent integration of models which are treated as black-box artifacts, so the model relationships are established between the interfaces that the models expose to the outside world. The composition is typically *sequential* or *parallel* (e.g., [5, 13, 17, 25]), and we focus on the latter [5], describing how it can be applied to components represented by state machines.

For example, assuming that MedicalRecord and Chart are the same objects in Figure 1, M2 and M4 can access and modify shared objects. Specifically, it is possible for a doctor and a nurse to both want to update an individual patient's Chart (MedicalRecord) simultaneously. Hence, M2 and M4 may need to interact, to synchronize on accesses to shared objects. To build a complete picture of this interaction and ensure that it is performed in a desired way, we need to create a composition of the scenarios in M2 and M4.

Figure 3 shows a simple component diagram, expressed in the visual modelling notation of the Darwin architectural description language [23]. The diagram describes the interconnections between three concurrent processes in the hospital system. These processes capture fragments of the scenarios in sequence diagrams M2 and M4 dealing with access to the patient's chart. C_1 is the process representing a doctor adding a prescription to a patient's chart (as shown by the update message in M2); C_2 is the process representing a nurse updating a patient's chart after a checkup (as shown by the update message in M4); and C_3 is the process representing a patient's chart which is accessed via a semaphore protocol.

The relationships between components in Figure 3 are defined between their interfaces, describing how the transition labels of each pair of components are mapped to one another. For example, the relationship between C_1 and C_3 is

acquire in C_1	maps to	lock in C_3
release in C_1	maps to	unlock in C_3

Similarly, the relationship between C_3 and C_2 is

acquire in C_2	maps to	lock in C_3
release in C_2	maps to	unlock in C_3

The behaviors of C_1–C_3, expressed in Labelled Transition Systems (LTSs) [25], are shown in Figure 4(a)–(c), respectively. To make sure that patients' charts are protected against concurrent changes, accesses to C_3 by C_1 and C_2 need to be mutually exclusive. This property can
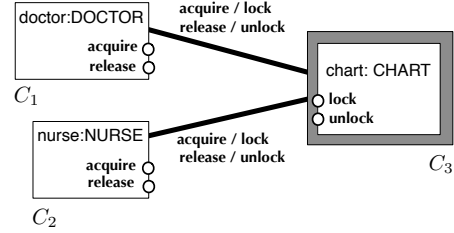

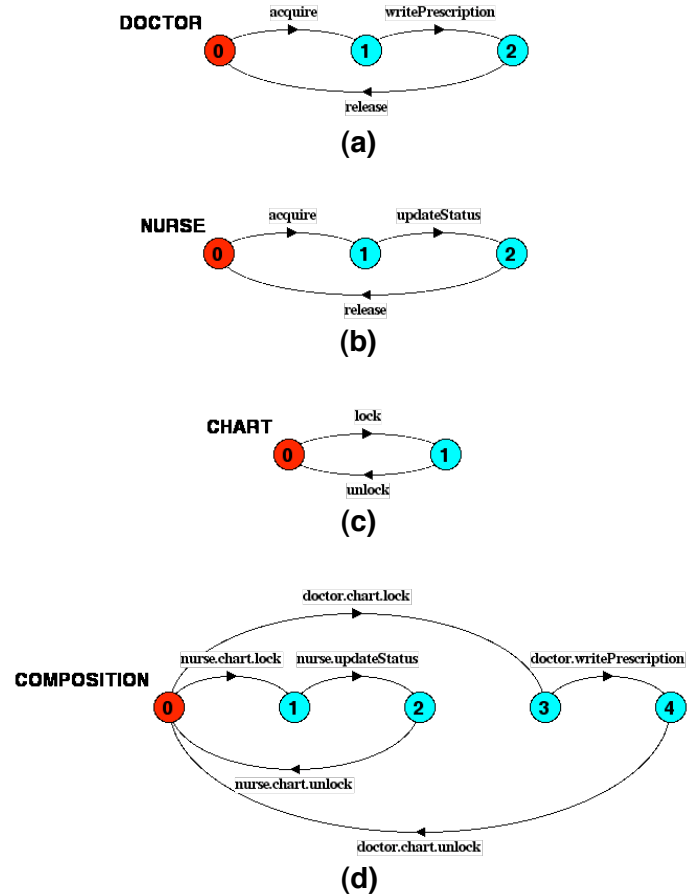**Figure 3. Interconnections between processes running in parallel.**

Figure 4. Specification of processes and their parallel composition.

be checked by computing the parallel composition of C_1–C_3 (see Figure 4(d)). In the composed model, an update made by the doctor is preceded by doctor.chart.lock (i.e., doctor's request to acquire access to the chart), and followed by doctor.chart.unlock (i.e., doctor's request to release the chart). The same is true for the nurse: her update is preceded by nurse.chart.lock and followed by nurse.chart.unlock. The checking of such properties is often automated. For example, we used LTSA [22] to express and compute the

4

parallel composition and to check it for mutual exclusion.

2.2 Weave

In Aspect-Oriented Software Development (AOSD), *weaving* is used to incorporate cross-cutting concerns into a base system. Weaving operators may be implemented in various ways depending on the nature of the base system and the concerns involved, and whether weaving is performed statically (at compile time) or dynamically (at runtime). Aspect-oriented languages usually include built-in constructs for weaving. For example, they provide point-cut constructs by which programmers specify where and when additional code (i.e., an aspect) should be executed in place of, or in addition to, an already-defined behavior (i.e., the base program). In aspect-oriented modelling, weaving is usually defined by patterns, selected either manually or automatically, using pattern matching techniques.

A classic example of a cross-cutting concern is logging, affecting all logged activities in a system. The perspective of the Computer Systems Administrator (model M7) in Figure 1 illustrates this concern for sequence diagrams. M7 is essentially a rewrite rule: if the left hand side of the rule (denoted by L) matches a fragment of a sequence diagram, then that fragment is rewritten using the right hand side of the rule (denoted by R). L matches if update*() (i.e., an operation whose name begins with the "update" prefix) is called on a persistent data object. If a match is found, the sequence diagram in question is modified as prescribed by R, resulting in SystemLog.append() being called after the update to append an entry to the system log.

The rewrite rule in M7 cross-cuts M2 and M4, as both M2 and M4 involve updates to persistent objects (MedicalRecord and Chart). Hence, it identifies the relationships between M7 and M2 and between M7 and M4. Specifically, the relation between M7 and M2 is

Object1 in M7	maps to	:Doctor in M2
Object2 in M7	maps to	:MedicalRecord in M2
update*() in M7	maps to	update() in M2

And the relationship between M7 and M4 is:

Object1 in M7	maps to	:Nurse in M4
Object2 in M7	maps to	:Chart in M4
update*() in M7	maps to	update() in M4

Figure 5 shows the result of weaving M7 into M2 and M4. In this example, weaving is performed statically, and the weaving operator can be most conveniently implemented using graph rewriting [42].

2.3 Merge

Merge is used to build a global view of a set of overlapping models that capture different perspectives on a certain functionality (e.g., [35, 28, 43, 8, 3, 7]). The goal of
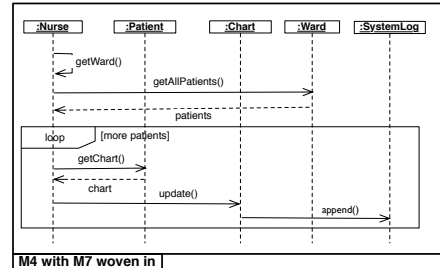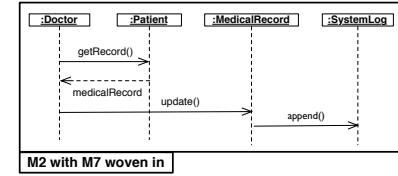

Figure 5. Weaving model M7 **into** M2 **and** M4**.**

model merging is to combine the input models by unifying their overlaps. In some cases, the overlapping aspects of the input models may be conflicting. Existing merging approaches differ in handling such cases: some approaches require that only consistent models be merged, implying that inconsistent models must be repaired prior to or during merge [20, 8]. Others tolerate inconsistencies by explicitly representing them in the resulting merged model (e.g., [7, 35, 28]).

To illustrate merge, consider models M1 and M3. M1 and M3 can overlap in several ways: the Patient class in M1 is likely to be the same as that in M3, and MedicalRecord in M1 is likely to be the same as Chart in M3. Consequently, (1) the update() operations of MedicalRecord and Chart are perhaps the same; (2) the aggregation between Patient and MedicalRecord in M1 is likely to be the same as the aggregation between Patient and Chart in M3; (3) the getRecord() and getChart() operations of Patient (in models M1 and M3, respectively) are likely to be the same as well. Assuming that all of these likely correspondences hold, the following is the mapping between M1 and M3:

Patient in M1	maps to	Patient in M3
MedicalRecord in M1	maps to	Chart in M3
Aggregation in M1	maps to	Aggregation in M3
update() in M1	maps to	update() in M3
getRecord() in M1	maps to	getChart() in M3

"Aggregation in M1" refers to the relation between Patient and MedicalRecord in M1, and "Aggregation in M3" refers to the relation between Patient and Chart in M3. Figure 6 shows the resulting merge of M1 and M3 with respect to the above overlap relationship between the models. As seen from the figure, the merge has only one copy of the common parts. When the source models have different vocabularies, we have two choices for naming the shared elements in the merge: either to define new terms, or to give preference to
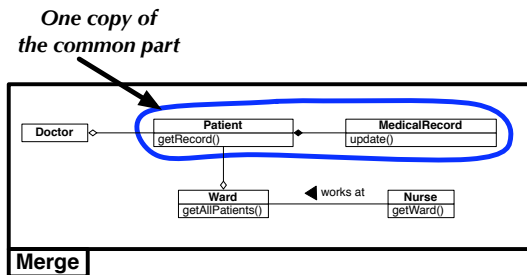
Figure 6. Merge of models M1 **and** M3**.**

the terminology in one of the source models. In the example of Figure 6, we chose to give terminological preference to the doctor's model (i.e., M1).

2.4 Important Considerations

Our exposition of composition, weaving and merge in earlier sections was meant to establish some general familiarity with these operators and how they are applied in practice. Yet, merely being explicit about the *type* of relationships between a set of models is insufficient for defining a new integration operator, or for choosing amongst a set of existing operators. In addition, we need to consider several other important factors, in particular:

1. What notation(s) are the source models expressed in?

2. What assumptions are made about the application context, the nature of the models, and their intended use?

3. What are the exact details of the relationships that need to be established (in terms of the level of granularity, semantics, representation)?

4. And, what quality and correctness criteria do we expect of the result of the integration?

In the past several years, we have been studying the merge operation in different domains. In the rest of this article, we elaborate on the factors outlined above, instantiating them to the model merging problem.

3 Model Merging Frameworks

We now focus on the merge activity and take a detailed look at the considerations, identified in Section 2.4, for designing a merge operator.

3.1 Merge Input

Merge operators are typically defined over a single notation [6]. This may necessitate some transformation over the source models, particularly when the source models are heterogeneous (i.e., represented in different notations), in which case they need to be translated into a common notation first. For example, merging a sequence diagram with a state machine may require a pre-processing translation of both models into a more detailed notation such as an LTS. Alternatively, the source models may be homogeneous (i.e., represented in the same notation), but it is advantageous to merge them after translation into an intermediate notation, e.g., to simplify the definition of the merge operator. For example, merge operators for hierarchical data definition languages such as nested-relational and XML schemas first translate the source models into flat schemas [24]. Another example is the resolution of parallelism in state machine merging, where parallel states are translated into their semantically equivalent non-parallel structures [28].

A key step in defining a merge operator is formalizing the notation for the input models, based either on their structure or their semantics. In a structural formalization, the models are often represented as graphs. This allows one to define generalizable merge solutions that apply to a variety of modeling languages, but this approach can complicate reasoning about the semantic properties of the merged model. In contrast, a semantics-based formalization of the models, e.g., trace-based, token-based, or tree-based [5], restricts the application of the developed merge operator to a specific modeling language, but provides a direct basis for reasoning about preservation of semantic properties during merge. In choosing between a graph-based versus a semantics-based formalization of merge, one must also consider the fact that level of formality and hence the emphasis on the semantics of the models increase as the development proceeds. Therefore, a graph-based approach is often more suitable in earlier stages of development, and a semantic-based one – in later stages.

Another factor that can affect how a merge operator is defined is the assumption that the developers make about the information that they do *not* explicitly capture in their individual models. For example, a model may intend to give a *closed-world* semantics, i.e., the missing information is regarded as false. In such a case, the merge operator needs to differentiate between the information discrepancies in the source models, e.g., by treating them as inconsistencies or as variations. Alternatively, a model may intend to give an *open-world* view, where the information not explicitly stated may still exist and be non-false. In such a case, the merge operator is additive, combining information available in the individual models. Examples of model merging under open- and closed-world assumptions are provided in Sections 4.1 and 4.2, respectively.
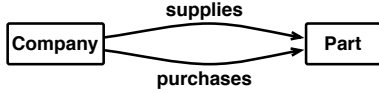
Figure 7. Associations with same endpoints.

3.2 Overlap Relationships

Before a merge can be applied, the exact nature of the overlap of the concepts between the two models needs to be explicated. Our experience indicates that often the analyst is not entirely sure about the exact nature of such an overlap, as there can be several reasonable alternatives. Each relationship should therefore be seen as a "hypothesis" about the overlap [35], and if a merge yields an unacceptable result, the reason might be because the models are genuinely inconsistent or because the correct nature of the overlap was not properly understood.

Table 2 lists some of the common ways in which concepts in different models may overlap. Each overlap is illustrated using models captured as graphs and as state machines with trace-based semantics [5]. We discuss the overlaps below:

Equivalence: An equivalence mapping between two elements means that they refer to exactly the same concept in the real world. For models with informal or semi-formal semantics, determining which concepts are equivalent is a choice to be made by the user because the precise meaning of the elements in the models relies heavily on the tacit beliefs, perceptions, and assumptions of the people who built them. In cell 1 of Table 2, the relationship states that the concept referred to as StaffMember in model P1 is the same as MedTeam-Member in model P2. Similarly, it indicates that Doctor in P1 and Physician in P2 are equivalent and so are the inheritance links between the concept pairs.

Even when the two endpoints are equivalent, links between them may not be so. For example, consider the model in Figure 7 which says that a Company plays a role of a supplier for some Parts and a role of a consumer for some others. Thus, we expect the equivalence of links between equivalent elements, such as the inheritance links between the concept pairs in models P1 and P2, to be determined manually.

In a semantic-based formalization, two elements are equivalent if they exhibit exactly the same semantic properties. In the case of state machines, states A and B are equivalent iff the behavior exhibited by A is exhibited by B and vice versa, meaning that A and B are *bisimilar* [25] – see the example in cell 2 of Table 2. Since bisimilarity relates states with *precisely* the same set of behaviors, this notion is often too strict for relat-

ing models with behavioural variations. Hence, a more flexible notion for capturing partial behavioural equivalence might be appropriate (see "similarity" discussed below).

Similarity: Two elements might be *similar* in some respects but not necessarily equivalent. For example, in cell 3 of Table 2, the concepts of Room (in P1) and Bay[1] (in P2) are determined to be similar. The point of similarity is that both Room and Bay are used for housing patients, but the two concepts are not equivalent as they differ in other respects, including patient amenities, available medical equipment, level of patient privacy, and staffing of medical care providers. In cell 3, we have used the "≈" notation to distinguish the similarity mappings from the equivalence mapping that relates Ward in P1 and Ward in P2.

Similarity relations are also very useful for relating models with formal semantics. Instead of considering pairs of states to be either bisimilar or dissimilar, one can introduce a quantitative similarity value for measuring how close the behaviors of one state are to those of another [28]. For example, in cell 4 of Table 2, states Resting and Idle have transitions to bisimilar states Ready and Responsive via identically labelled transitions, but they are not equivalent as their behaviors differ on the transition labelled to Patient Rm. The quantitative similarity value, adapted from [39], yields a number between 0 and 1, and when this number is above a given threshold, the states are considered to be similar.

Two useful heuristics that one can use to distinguish similarity from equivalence relations are as follows:

1. As we explain in Section 3.3, a criterion for merge is to not have duplicate elements (non-redundancy). Therefore, if two elements a and b are related via an equivalence relation, they will be unified and collapsed into a single element by the merge. If this outcome is intended, an equivalence relation should be used. Otherwise, a similarity relation should be considered, provided that more specific types of overlaps, e.g., generalization or aggregation elaborated later in this section, do not apply.

2. Equivalence is a transitive notion, whereas similarity is not. Specifically, $a = b$ and $b = c$ implies that $a = c$. In contrast, $a \approx b$ and $b \approx c$ does not imply $a \approx c$. For a simple example, the cities of Vancouver and Toronto may be deemed similar as both are metropolitan cities

---

[1] A bay at a hospital is an area where multiple patient beds are set up, often with the beds separated from one another using partitions.

Table 2. Different types of overlapping relationships.

Overlap Type	Graph-Based (Class Diagrams)	Semantics-Based (State Machines)
Equivalence	StaffMember, Nurse, Doctor (P1); MedTeamMember, Physician, Technician (P2) — **1**	Resting, to Ward, Responsive (P1); Idle, to Ward, Ready (P2) — **2**
Similarity	Ward, Room (P1); Ward, Bay (P2); ≈ — **3**	Resting, to Ward, Responsive (P1); Idle, to Ward, to Patient Rm, Ready, In Patient's Room (P2) — **4**
Generalization	Critical Care Unit (P1); Ward, Special Unit (P2) — **5**	Ready, In Patient's Room (P1); Ready, Bath Change, Blood Pressure (P2) — **6**
Aggregation	Equipment, Bed, PatientMonitor (P1); Ward, Room (P2) — **7**	Not Applicable — **8**
Overriding	InpatientVisit Record, OutpatientVisit Record (P1); ✖Inpatient, ✖Outpatient (P2) — **9**	Resting, to Patient Rm, In Patient's Room (P1); Resting, ✖ to Ward, In Patient's Room (P2) — **10**
Information Gap	CriticalCare Unit (P1); SurgeryUnit (P2); SpecialUnit, CriticalCare Unit, Surgery Unit (P3) — **11**	Resting, to Ward, Ready (P1); In Patient's Room, to Common Rm, Idle (P2); Ready, to Patient Rm, In Patient's Room (P3) — **12**

(Vancouver ≈ Toronto). Toronto and Brampton too may be deemed similar in that they are in the same province, are closely located, and offer access to virtually the same amenities and services (Toronto ≈ Brampton). However, we cannot conclude from the above that Vancouver ≈ Brampton, because the link, Toronto, overlaps with Vancouver and Brampton in different ways. This distinction leads to a second heuristic: For a and b to be related via equivalence, it is necessary that

any relationship that any element x has towards a should transit over to b, and vice versa. If this is not the case, a similarity relation should be considered, again provided that one cannot be more specific about the type of overlap.

Generalization: An element in one model can be a generalization or specialization of elements in another model. For example, in cell 5 of Table 2, Special Unit is a generalization of Critical Care Unit because a Hospital's special unit can be either a critical care unit or a

surgery unit.

Generalization can also be used in a semantic-based domain. For state machines, the behavior of a state in one model can refine (generalizes) the behaviors of other states in another model. In the example in cell 6 of Table 2, the trace in model P2 describes the detailed steps that a nurse should follow during the daily control routine: first doing a bath change and then checking the blood pressure. The trace in model P1 abstracts these steps via a single state, In Patient's Room. Hence, the trace in model P1 is more general than the trace in P2, with state In Patient's Room refining the sequence Bath Change and Blood Pressure, and the relationship between the two models is behavioural generalization.

Aggregation: A pair of elements can be related through a has-a or is-part-of relationship. For example, in cell 7 of Table 2, Room is declared as having Equipment pieces as sub-parts. Aggregation relationships are most useful for capturing the structural decomposition of the elements in a domain. Such decomposition does not apply to state machine models.

Overriding: Another interesting relationship is *retrenchment*, or *override*, which allows developers to withdraw from their positions as their knowledge evolves or to avoid inconsistency with other developers. For example, when comparing P1 and P2 in cell 9 of Table 2, we determine that Inpatient and Outpatient are ill-defined concepts: these characteristics are determined by the nature of a patient's visit to the hospital rather than being invariantly associated with a patient. Therefore, we may want to refute the concepts Inpatient and Outpatient in model P2 and override (replace) them with InpatientVisitRecord and OutpatientVisitRecord of model P1, respectively. In Table 2, refuted information is denoted by the ✗ symbol.

Override relationships are applicable to the semantic-based case as well. For example, the state machine P1 in cell 10 of Table 2 indicates that a nurse may go from the nurses' resting area to a patient's room whenever the patient needs help. But according to the state machine P2, nurses that are resting do not respond to patients' requests as the transition from the Resting state to the In Patient's Room state is marked as refuted (✗). Thus, the two models disagree on whether nurses in the resting area should respond to patients' requests, which is captured by the override relationship between the behavior of P1 and P2. This relationship, unlike others, is not monotonic – the resulting merge will not necessarily have more information than both of the original models.

Information Gaps: Elements in different models can have more complex relationships than those described above. In particular, there may be information gaps (discontinuities) between the source models that need to be bridged first, before meaningful relationships can be defined. For example, in cell 11 of Table 2, to relate CriticalCareUnit and SurgeryUnit, we first need to introduce a more abstract concept, such as SpecialUnit, and then declare CriticalCareUnit and SurgeryUnit to be specializations of that concept. Another example of information gaps is when a concept in one model is derived from concepts in other models through a computation. For example, one model may have a single attribute, name, for the full name of StaffMember, while another model may have two attributes, firstName and lastName, for the same purpose. To relate these two models, we need to account for the fact that one's full name is computed by concatenating their first and last names.

Information gaps can exist in the semantic-based case as well. For state machines, information gaps relations can be used to combine pieces of behavior from different models to generate a more complete description. For example, cell 12 in Table 2 presents a case where the behaviors in models P1 and P2 are connected through the sub-trace in P3. Specifically, by concatenating the behaviors in these models, we obtain a full behavioural description indicating that a nurse can move from her resting area to the ward where she may go to patients' room to respond to their requests and can return to the resting area afterwards.

3.3 Desirable Criteria for Merge

Depending on the goals to be achieved by model merging, e.g., allowing inspections or consistency checking, the merge result may be expected to meet various criteria. Most common of these are given below.

- **Completeness**: If a concept appears in one of the source models, it is represented in the merged model as well [1]. This is to ensure that no information is lost in the merge process.

- **Non-redundancy**: If a concept appears in more than one source model, only one copy of it is included in the merged model [1].

- **Minimality**: Merge does not introduce new information, which is neither present nor implied by the source models.

- **Totality**: Merge is computable for an arbitrary set of models. This property is of particular importance if one wants to tolerate inconsistency between the source models [32, 30].

- **Soundness**: Merge supports the expression and preservation of semantic properties. For example, if models are expressed as state machines, one may want to preserve their behaviors, expressed as temporal logic properties, to ensure that the intended meaning of the source models is properly captured in the merge. If soundness is required, a merge framework needs a capability to *express* desired properties and *guarantee* that they are preserved.

Of course, not all of the above criteria may be desirable, or even applicable, to all merge operators. For example, completeness and minimality may be undesirable if model merging also involves conflict resolution, in which case the final merge can potentially add or delete information [31]. Semantic preservation may be undesirable when one wants to induce design drift or perform an abstraction during merge, since such manipulations are usually not semantics-preserving [18]. Finally, totality may be undesirable when the source models are expected to seamlessly fit together. In such a case, the source models should be made consistent before they are merged [8].

In Section 4, we describe two different merge operators developed in our earlier work in response to different needs and in different application domains. Our discussion is guided by and structured around the range of considerations outlined in the current section.

4 Two Example Merge Operators

We describe two instantiations of the merge operator: *Algebraic Merge* (Section 4.1) and *State Machine Merge* (Section 4.2). We apply both operators to models M5 and M6 in Figure 1. In model M5, a nurse tends to a patient in a state In Patient's Room. This perspective also includes a transition between Resting and In Patient's Room modeling the fact that a nurse can move from the nurses' common room to a patient's room without needing to register herself in a ward as an available nurse. Model M6 distinguishes two separate activities that a nurse may carry out while in a patient's room: Check Temperature and Blood Pressure and Bath Change. This perspective also assumes that a nurse can tend to a patient only when she is registered in a specific ward, i.e., there is no transition from state Available to either of the above two states. Finally, M6 includes a nurse going Off Duty when the shift changes. She can then remain in the common room but cannot go to the ward or to a patient's room until she becomes available during the next shift change.

For each of the merges, we present the underlying assumptions, using the framework presented in Section 3.

4.1 Algebraic Merge

Our algebraic merge is a generic operator that works over graph-based models. Relationships between models are captured by sub-graphs, also referred to as *connectors*. The outcome of merge is characterized by an algebraic concept called *colimit* – an operation for gluing objects together with nothing essentially new added and nothing left out [9].

Algebraic merge is typically used during early phases of development where the most important goals are exploring the ontological relationships between the terms used in different models, aligning the conceptual structures of different stakeholders, and producing an abstract perspective of the entire system being developed. We now illustrate the steps of algebraic merge of models M5 and M6 in Figure 1. In the first step, the user creates a connector model, which includes just the overlaps between M5 and M6, and specifies how the connector maps onto each of the two models. Figure 8 shows the *equivalence* (cell 1 of Table 2) mappings between the connector model and M5 and M6.

The second step is to compute the merge of M5 and M6 with respect to their overlaps as described by the connector. The result is shown in Figure 9. As can be seen from the figure, algebraic merge unifies into a single element any set of source model elements that have been mapped through the connector. For example, the Ready states in M5 and M6 are unified into a single state in the merge because they are mapped through the connector model. More interestingly, states Bath Change and Check Blood Pressure and Temperature, both from M6, and In Patient's Room from M5 are all unified into a single state of the merged model (In Patient's Room). As we describe in Section 4.2, our state machine merge operator differs from our algebraic merge operator in a number of respects, most notably, in that it never collapses different states of the same model into a single state.

Using colimits ensures that the syntactic structure of the source models is maintained in their merge. Specifically, for any source model M and any graph edge r from node x to y (denoted $r : x \rightarrow y$) in M, if r is represented in the merge by $r' : x' \rightarrow y'$, then x and y must be represented by x' and y', respectively. For example, the transition to Ward : Available $\rightarrow$ Ready in M6 is represented by to Ward : Resting $\rightarrow$ Ready in the merge. Colimits then ensure that Available and Ready in M6 are respectively represented by Resting and Ready in the merge.

Table 3 summarizes the main characteristics of algebraic merge with regard to the factors described in Section 3. We discuss the columns of the table below.

Input Models. The operator applies to homogeneous graph-based models and takes an open-world (additive) approach for handling the information that is not explicitly captured in individual models. For example, the shift change scenario is present only in model M6 and not M5,

Table 3. Characteristics of algebraic and state machine merge operators.

Merge operators	Input Models	Relationships	Soundness	Properties
Algebraic Merge	Homogeneous, graph-based Open-world assumption	Applies to all overlap relations in Table 2	Preserves positive existential LFP properties and one level of universal quantification	Complete, Non-Redundant, Minimal, Total
State Machine Merge	Homogeneous, state machines with trace-based semantics Closed-world assumption	Applies to the overlap relations (2), (4), (6), and (12) in Table 2	Preserves all LTL properties	Complete, Total

and yet it is included in the resulting merge. Similarly, the merge also includes the behavior that enables the nurse to go straight from the common room to a patient's room – this behavior is expressed only in M5.

Relationships. Algebraic merge has the flexibility to incorporate all notions of overlap described in Table 2. For instance, in the example of Figure 8, we could choose not to equate states Bath Change and Check Temperature and Blood Pressure in M6 with state In Patient's Room in M5. Instead, we could use similarity mappings to say that Bath Change and Check Temperature and Blood Pressure are similar but not equivalent to In Patient's Room. With such similarity mappings, we would obtain the merged model shown in Figure 10. In this merge, the three states in question and their incident transitions are kept distinct, and the similarity relations between the states are explicitly recorded in the merge. This is in contrast to the merge of Figure 9, where the three states collapsed into a single one, and the incident transitions with a common label were unified.

Soundness. As we discussed earlier, colimits ensure that the syntactic structure of the source models is preserved in the merge. However, preservation of syntax does not allow one to directly argue about the preservation of semantics, for which we typically need to reason about logical properties. Specifically, given a property φ expressed in a particular logic, one cannot readily determine from the definition of colimit whether φ carries over from the source models to the merged model. In [36], we use techniques from finite model theory [21] to enable such type of reasoning for colimits. Specifically, we show that colimits preserve a fragment of Least Fixpoint Logic (LFP) – the extension of First Order Logic with a least fixpoint operator. The preserved fragment of LFP consists of all positive existential properties (i.e., properties with no negation or universal quantification) plus one level of universal quantification over positive existential properties (i.e., properties of the form $\forall x.\, \varphi(x)$, where φ is positive existential).

For state machines, an interesting consequence of these preservation results is the preservation of *reachability*. For example, knowing that there is a path from Available to Bath Change in M6, we can guarantee that there is path from Resting to In Patient's Room in the merges shown in Figures 9 and 10. For a detailed discussion about the connection between the preserved fragment of LFP and some common soundness constraints, see [36].

Properties. The use of colimits for merge ensures that the outcome is complete (in the sense that it fully contains all the source models), minimal, redundancy-free, and total. In addition, using colimits has the advantage that the merge operator directly applies to *systems*, rather than *pairs*, of models [35]. That is, the operator works for any number of models and relationships.

4.2 State Machine Merging

Our second merge operator is specifically aimed at state machine models and is motivated by the need to preserve the semantic properties of these models in their merges. These properties are typically expressed in variants of temporal logic formulas. Overlaps between state machines are specified using binary relations over their state spaces. The merge is defined as a *common refinement* of a set of state machines with respect to their relationships [19]. Basing the notion of state machine merge on refinement is standard [41, 15]. Refinement captures the process of combining behaviors of individual models while preserving all of their agreements. This guarantees that semantic properties of the source models carry over to their merge.

State machine merge operator is more suitable for late stages of development where the goal is to obtain a sound and operational model of the system under development. In this kind of merge, it is assumed that all "real" disagreements between stakeholders have been resolved, and thus the remaining discrepancies between the behaviors of the
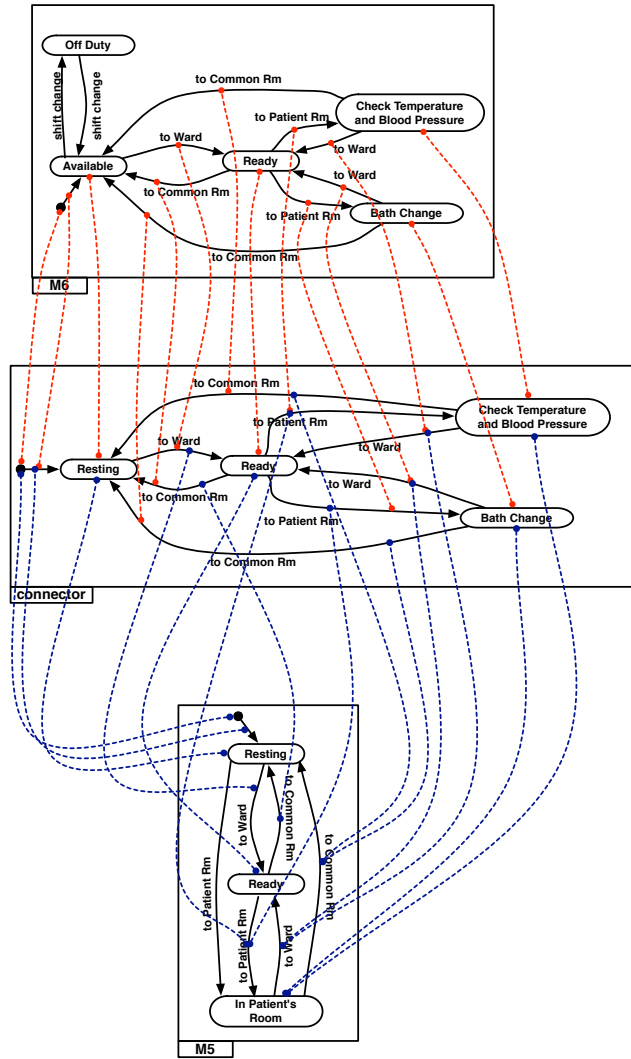
Figure 8. Mappings between the connector model and models M5 **and** M6**.**

Figure 9. Algebraic merge of state machines M5 **and** M6 **with respect to the mappings in Figure 8.**

input models can be treated as variabilities in the models intended functionality. These variabilities are represented as conditional behaviors in the merge [28].

Below, we illustrate the steps of state machine merging. First, the user identifies a relationship between the input state machines using a binary relation over their states. Figure 11 shows a mapping between the states of the input state machines M5 and M6. Unlike the algebraic merge, mappings in state machine merge are limited only to states. The reason is that bisimilarity does not distinguish between multiple parallel transitions with the same label, always treating them as a single transition. Therefore, to indicate mappings between transitions, it is sufficient to map their endpoints and assume that the transitions with similar labels are similar.

The next step is to compute the merge with respect to

the binary relation defined above. The result is shown in Figure 12. As can be seen from the figure, non-shared behaviors are guarded by conditions denoting the originating view that exhibits those behaviors. Further, unlike the algebraic merge, state machine merge does not collapse distinct states. For example, the states Check Temperature and Blood Pressure and Bath Change of model M6 remain intact in the merge, although these two states are mapped to a single state in model M5. This is necessary for preserving the common behaviors of the input state machines in their merge [28]. For example, the property

Prop$_1$:"whenever the nurse is in ward, she can tend to a patient"

holds in both models M5 and M6 and is preserved in their merge. However, the property

Prop$_2$: "a nurse not registered in a ward can tend to a patient"

which holds in model M5 but not in M6, is represented as a parameterized behavior in the merge and is preserved only when the guard **[in M5]** holds. This guard indicates the origin of the transition and becomes true only when the assumptions under which M5 is applicable hold.

Table 3 summarizes the main characteristics of the state machine merge with regard to the factors described in Section 3. The columns of the table are discussed below:

Input Models. The operator applies to state machine models and takes a closed-world approach for handling the information that is not explicitly captured in individual models. For example, the shift change process which is modelled only in M6, and the behavior in M5 that enables the nurse to go straight from the common room to a patient's room are both included in the merge in Figure 12, but as parameterized behaviors, indicating that these have not been
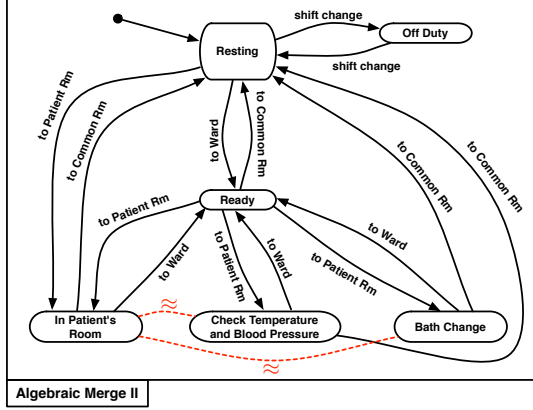
Figure 10. Algebraic merge of state machines M5 **and** M6 **with respect to the mappings in Figure 8 but with** In Patient's Room **related to** Bath Change **and** Check Temperature and Blood Pressure **via similarity relations.**
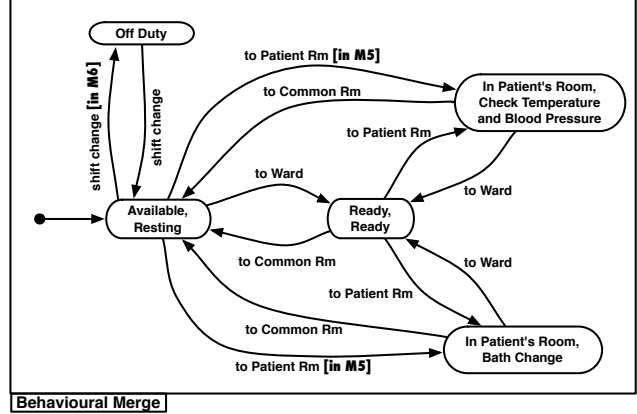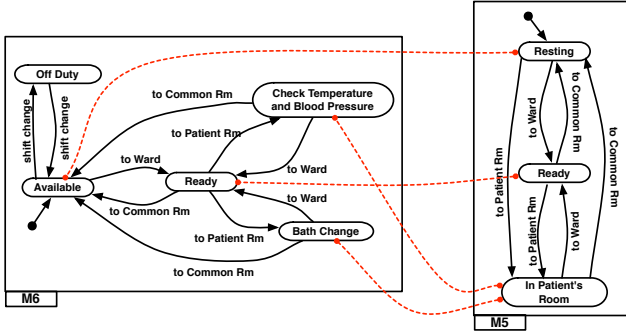

Figure 11. Binary relation between the state machines M5 **and** M6.

present in both input models.

Relationships. The state machine merge can incorporate four notions of overlap presented in Table 2, namely, equivalence, similarity, generalization, and override. Aggregation is not applicable to the state machine notation. Information gap requires a merge operator that can concatenate behaviors, potentially generating new behaviors in the merge – those that did not come from either of the input models. This operator is principally different from the state machine merge described here.

Soundness. The state machine merge operator is sound [29] for linear time temporal logic properties (LTL) [33]. That is, true behavior properties of the models expressed as LTL formulas also hold in their merge. For example, both $Prop_1$ and $Prop_2$ discussed above can be expressed as LTL formulas, and the state machine merge operator preserves both of them: $Prop_1$, which refers to shared behavior between M5

Figure 12. State machine merge of the state machines M5 **and** M6 **with respect to the mapping in Figure 11.**

and M6, is preserved as a non-parameterized behavior in the merge, and $Prop_2$, which refers to a non-shared behavior, is preserved as parameterized one. In short, the state machine merge includes, in either guarded or unguarded form, every behavior of the input models. The use of parameterization for representing behavioural variabilities allows us to generate behavior-preserving merges for models that may even be inconsistent.

Properties. Finally, state machine merge is always complete, since it preserves all behaviors of the input models. It is also minimal and total, i.e., information not present in the input models does not appear in the merge, and further, this merge operator can be applied to every pair of state machines. Non-redundancy property, however, may not hold. For example, the states Check Temperature and Blood Pressure and Bath Change of model M6 remain separate in the merge even though they are mapped to the same state in model M5.

In this section, we used the merge framework in Section 3 to compare two different merge operators: algebraic and state machine. We applied these operators to the same example to show how the same models can be merged in completely different ways when the factors triggering the need for merge are different. The state machine merge, while it allows for careful analysis of semantic properties of the merged model, is applicable only to state machine-based notations. The algebraic merge, however, is applicable to any graphical notation.

5 Tool Support

We have implemented the two merge operators described in Section 4 as part of a tool called TReMer+ [37]. Automated assistance for matching the elements of different models is available for our state machine merge operator
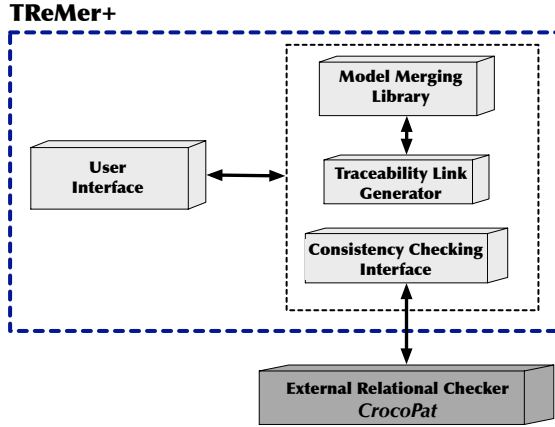
13

Figure 13. Architecture of TReMer+.

as an external tool described in [28]. The architecture of TReMer+ is shown in Figure 13. Its main blocks are as follows:

User Interface. TReMer+ provides a visual user interface for editing models, building relationships between models, and defining systems made up of multiple models and relationships. TReMer+ currently supports entity-relationship diagrams, state machines, and simple UML domain models. In the future, we plan to extend the tool to support other notations, such as goal models and detailed class diagrams.

Merge Library. TReMer+ defines a plugin interface for the merge operation and can work with any merge algorithm that realizes this interface. Currently, we provide implementations for the two merge algorithms discussed in Section 4.

Traceability Link Generator. TReMer+ provides facilities for maintaining traceability between the merged models and their sources. The traceability links are generated automatically during the merge computation and can be accessed through the user interface for easy navigation from a merged model back to its source models and relationships.

Consistency Checking Interface. TReMer+ provides support for checking the consistency of the relationships and the merged models. Our consistency checking approach, described in [38], uses an existing relational checker, CrocoPat [2], for verification of consistency properties expressed in first order logic. TReMer+ interacts with CrocoPat through an interface responsible for (1) translating graphical models into CrocoPat's predicate language; (2) invoking CrocoPat with a user-defined set of consistency rules; and (3)

communicating the inconsistency diagnostics generated by CrocoPat to the user interface for presentation to the user.

TReMer+ is written in Java and uses around 15,000 lines of code, of which 8,500 implement the user interface, 5,500 implement the tool's core functions (model merging, traceability, and serialization), and 1000 implement the interface for interacting with CrocoPat. The most recent version of the tool along with the material used in our previous studies with the tool are freely available at `http://se.cs.toronto.edu/index.php/TReMer+`.

TReMer+ has been applied in two real case studies, both dealing with independently-developed models. The first study, used for evaluating our algebraic merge operator, included a set of UML domain models for a health care system developed by the students of an advanced undergraduate course on object-oriented modelling. These models were roughly equal in size, each with 60 to 70 elements (nodes and edges), but with remarkable discrepancies in the way the models were structured. The second study, used for evaluating our state machine merge operator, was based on a set of variant telecommunication features from AT&T. These features were expressed as Statecharts [10] and had 50 to 90 elements (states and transitions).

These two studies were motivated by different research questions relevant to the context and purpose of each study. In the first study, the main question was whether algebraic merge was useful for aligning the vocabularies and structures of different models and arriving at a unified, consistent domain model. In the second study, the main question pursued was whether state machine merge could facilitate the maintenance of variant features by constructing a single parameterized model. TReMer+ was applied successfully in both studies for construction of models and relationships, generating merged models, visualizing and inspecting the results, and automatic consistency checking. For full details of the first study, see [34]. For the second, see [27].

6 Discussion and Conclusion

In this article, we argued that complexity of integration of a set of models can be reduced by explicating the type of relationships which hold between these models. Specifically, we showed that application of key integration activities studied in the literature – merge, composition and weaving – is completely induced by different relationship types, thus giving us reasons to believe that relationships should be treated as first-class artifacts during the integration process. Yet, even when the choice of the integration activity is made, there can be a variety of considerations as to which particular version of the operator is most applicable. We described factors that determine the definition

of the merge operator: details of relationships, assumptions on input models, and desired soundness properties of the merged model. We illustrated the flexibility and generality of the resulting merge framework by systematically comparing two very different merge operators developed in our previous work.

We believe that results of this article would provide a useful guide for the development of new model integration operators. Particularly, with the recent interest in Global Software Engineering (GSE) [14], there is now an increasing demand for flexible integration techniques for consolidating models that are built by geographically distributed teams. We anticipate GSE to spur new research on model integration in general, and model merging in particular. This underscores the importance of developing conceptual frameworks, similar to the one we developed in this article, for classifying and systematizing the research on model integration.

There are several topics that provide fruitful directions for future research. With regards to model merging, it is important to develop a more thorough classification of the purposes for which merge is applied, and subsequently study the applicability and tradeoffs between different merge operators for a given purpose. Particularly, there is a tradeoff between (1) the degree of flexibility in the specification of inter-model relationships and (2) the degree to which logical soundness constraints are preserved by merge. For example, for merging state machines, the algebraic merge operator in Section 4.1 offers a lot of flexibility in terms of the types of overlap relationships used, but at the cost of providing only limited guarantees on soundness. In contrast, the merge operator in Section 4.2 supports fewer types of overlap relationships; but the operator provides strong guarantees on soundness. For developers to be able to choose between different merge operators with ease, a clearer mapping needs to be developed between the purpose of merge and the types of soundness guarantees required for that purpose.

Improving tool support for merge also warrants future research. Specifically, existing modelling platforms, e.g., the Rational Software Architect [16], are primarily aimed at centralized development, where all developers contribute to a single holistic model. These platforms lack support for important distributed development activities such as constructing explicit relationships between models, and defining and navigating systems of inter-related models. We intend to build on the experience gained through the development of our prototype merge tool (Section 5), and augment popular modeling tools such as RSA with features for facilitating distributed development.

Another important topic for future work is to study how different integration operators can be built to work together. In particular, we would like to investigate how one can combine a collection of models that are interconnected via not just one but multiple relationship types, i.e., the situation where some models are overlapping, some models are interacting, and some models are cross-cutting others. Further, we plan to extend our conceptual framework to encompass other model management operations such as consistency checking and change propagation [3, 4].

References

[1] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[2] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, 2005.

[3] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *Workshop on Global Integrated Model Management (GaMMa'06) co-located with ICSE'06*, 2006.

[4] M. Chechik, W. Lai, S. Nejati, J. Cabod, Z. Diskin, S. Easterbrook, M. Sabetzadeh, and R. Salay. Relationship-based change propagation: A case study. In *ICSE Workshop on Modeling in Software Engineering (MiSE'09)*, 2009.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[6] P. Darke and G. Shanks. Stakeholder viewpoints in requirements definition: A framework for understanding viewpoint development approaches. *Requirements Engineering*, 1(2):88–105, 1996.

[7] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *ICSE'01: Proceedings of the 23rd International Conference on Software Engineering*, pages 411–420, 2001.

[8] D. Fischbein, G. Brunet, N. D'Ippolito, M. Chechik, and S. Uchitel. Weak alphabet merging of partial behaviour models. *ACM Transactions on Software Engineering and Methodology*, 2010. To appear.

[9] J. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[10] D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw Hill, 1998.

[11] W. Harrison, H. Ossher, and P. Tarr. General composition of software artifacts. In *5th International Symposium Software Composition (SC'06), co-located with ETAPS'06*, volume 4089 of *Lecture Notes in Computer Science*, pages 194–210. Springer, 2006.

[12] W. Harrison, H. Ossher, P. Tarr, V. Kruskal, and F. Tip. CAT: a toolkit for assembling concerns. Technical Report RC22686, IBM Research, 2002.

[13] J. Hay and J. Atlee. Composing features and resolving interactions. In *SIGSOFT'00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 110–119, 2000.

[14] J. Herbsleb. Global Software Engineering: The Future of Socio-Technical Coordination. In *Future of Software Engineering Track of the 29th International Conference on Software Engineering*, pages 188–198, 2007.

[15] A. Hussain and M. Huth. On model checking multiple hybrid views. In *Proceedings of 1st International Symposium on Leveraging Applications of Formal Methods*, pages 235–242, 2004.

[16] IBM Rational Software Architect. `http://www.ibm.com/software/awdtools/architect/swarchitect/`.

[17] M. Jackson and P. Zave. Distributed feature composition: a virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, 1998.

[18] Y. Kalfoglou and M. Schorlemmer. Ontology mapping: The state of the art. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold, editors, *Semantic Interoperability and Integration*, number 04391 in Dagstuhl Seminar Proceedings. IBFI, 2005.

[19] K. Larsen and B. Thomsen. A modal process logic. In *LICS'88: Proceedings of 3rd Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE Computer Society Press, 1988.

[20] K. Letkeman. Comparing and merging UML models in IBM rational software architect. Technical report, IBM, 2006.

[21] L. Libkin. *Elements Of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[22] J. Magee and J. Kramer. *Concurrency: State models and Java Programming: 2nd Edition*. Wiley, 2006.

[23] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the behaviour of distributed software architectures: A case study. In *Proceedings of 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 240–247, 1997.

[24] S. Melnik, E. Rahm, and P. Bernstein. Rondo: a programming platform for generic model management. In *SIGMOD'03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 193–204, 2003.

[25] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[26] A. Moreira, A. Rashid, and J. Araújo. Multi-dimensional separation of concerns in requirements engineering. In *RE'05: Proceedings of the 10th IEEE International Symposium on Requirements Engineering*, pages 285–296, 2005.

[27] S. Nejati. *Behavioural Model Fusion*. PhD thesis, University of Toronto, 2008.

[28] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of Statechart specifications. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 54–64, 2007.

[29] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of variant feature specifications. Submitted for publication, 2010.

[30] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, 2003.

[31] N. Noy and M. Musen. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 450–455, 2000.

[32] B. Nuseibeh, S. Easterbrook, and A. Russo. Making inconsistency respectable in software development. *The Journal of Systems and Software*, 58(2):171–180, 2001.

[33] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[34] M. Sabetzadeh. *Merging and Consistency Checking of Distributed Models*. PhD thesis, University of Toronto, 2008.

[35] M. Sabetzadeh and S. Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requirements Engineering Journal*, 11(3):174–193, 2006.

[36] M. Sabetzadeh, S. Nejati, M. Chechik, and S. Easterbrook. Reasoning about consistency in model merging. In *Workshop on Living With Inconsistency in Software Development (LWI'10) co-located with ASE'10*, 2010.

[37] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. Global consistency checking of distributed models with TReMer+. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 815–818, 2008.

[38] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *RE'07: Proceedings of 15th IEEE International Requirements Engineering Conference*, pages 221–230, 2007.

[39] O. Sokolsky, S. Kannan, and I. Lee. Simulation-based graph similarity. In *TACAS'06: Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2006.

[40] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE'99: Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, 1999.

[41] S. Uchitel and M. Chechik. Merging partial behavioural models. In *SIGSOFT'04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 43–52, 2004.

[42] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi. An expressive aspect composition language for UML state diagrams. In *MoDELS'07: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, pages 514–528, 2007.

[43] J. Whittle and J. Schumann. Generating Statechart designs from scenarios. In *ICSE'00: Proceedings of 22nd International Conference on Software Engineering*, pages 314–323. ACM Press, May 2000.