Scalable Multiple-View Analysis of Reactive Systems via Bidirectional Model Transformations

Christos Tsigkanos
TU Wien
Vienna, Austria

Nianyu Li
Peking University
Beijing, China

Zhi Jin
Peking University
Beijing, China

Zhenjiang Hu
Peking University
Beijing, China

Carlo Ghezzi
Politecnico di Milano
Milano, Italy

ABSTRACT

Systematic model-driven design and early validation enable engineers to verify that a reactive system does not violate its requirements before actually implementing it. Requirements may come from multiple stakeholders, who are often concerned with different facets – design typically involves different experts having different concerns and views of the system. Engineers start from a specification which may be sourced from some domain model, while validation is often done on state-transition structures that support model checking. Two computationally expensive steps may work against scalability: transformation from specification to state-transition structures, and model checking. We propose a technique that makes the former efficient and also makes the resulting transition systems small enough to be efficiently verified. The technique automatically projects the specification into submodels depending on a property sought to be evaluated, which captures some stakeholder's viewpoint. The resulting reactive system submodel is then transformed into a state-transition structure and verified. The technique achieves cone-of-influence reduction, by slicing at the specification model level. Submodels are analysis-equivalent to the corresponding full model. If stakeholders propose a change to a submodel based on their own view, changes are automatically propagated to the specification model and other views affected. Automated reflection is achieved thanks to bidirectional model transformations, ensuring correctness. We cast our proposal in the context of graph-based reactive systems whose dynamics is described by rewriting rules. We demonstrate our view-based framework in practice on a case study within cyber-physical systems.

ACM Reference Format:
Christos Tsigkanos, Nianyu Li, Zhi Jin, Zhenjiang Hu, and Carlo Ghezzi. 2020. Scalable Multiple-View Analysis of Reactive Systems via Bidirectional Model Transformations. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Australia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ASE 2020, 21 - 25 September, 2020, Melbourne, Australia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Design complexity for a system is high because of the many different intertwined facets that need to be taken into account. Furthermore, different viewpoints (or system aspects) typically need to be accommodated [37]. Different stakeholders have different concerns, and different experts may be involved in the design of the system, focusing on different technical aspects. For example, a distributed system may be comprised of different software components connected through networks – the arrangement of components in a software architecture is one viewpoint, while the network they form is another. The advisable way to dominate complexity is to apply sound software engineering principles–namely separation of concerns and abstraction–to system design and early validation.

We consider the case where requirements to be satisfied by a reactive system under design are specified in terms of assertions in a temporal logic and that the system model is encoded into a formal graph-based modeling language. To support early design validation, the behaviors that can emerge within the reactive system need to be verified against the requirements. This verification can be supported by model checking, which requires an interpretation of the reactive system specification as a state machine. Model checking does an exhaustive search of the state space for absence of illegal behaviors [8]. However, like any exhaustive technique, such state machine interpretation and its model checking may become impractical as models become large and complex.

To address these difficulties, this paper investigates a technique that leverages the different properties expressing the different concerns of stakeholders to automatically project the specification model into submodels, each depending on the property under consideration. Projections automatically generate submodels that are equivalent to the full source model from the standpoint of the specific properties that need to be verified. Projected models, however, are generally smaller than the full source model, and therefore verification via model checking, which may be unfeasible for the full reactive system model, may become feasible for the submodel. For example, to reason about the deployment of components in a software architecture, we only need to care about structure and connectivity of the infrastructure on which the different components of an application is hosted, and not e.g., network aspects.

Once a submodel is derived, a designer who is concerned with that specific view may analyze it and correct it if problems are found. For example, to improve performance, a database expert may decide that a cache should be placed in front of a database within a virtual machine, reflecting that in the deployment model.

In our proposal, we assure that changes applied to the submodel are reflected back automatically in the full model, and from there propagated to other projections [29] that may also be affected and hence would need to be re-validated. The full reactive system specification model and the various submodels for the different views are therefore automatically kept in sync, supporting separation of concerns. Automated reflection is achieved thanks to the use of bidirectional model transformations [11]. In our example, we would need to verify that adding the new database cache does not lead to violation of other requirements (for example referring to database network connectivity), which pertain to another view of the system represented by a different submodel.

We leverage results on model slicing [3][1] and devise a practical software engineering framework that can support multiple-view analyses, keeping views automatically synchronized. The slicing process [9, 19] is performed at the reactive system specification level, leveraging the types that appear in the specification and in the properties under consideration. Our technique achieves scalability in analysis in two dimensions: (i) slicing is performed at the reactive system specification level, before the (expensive) generation of the state-transition structure describing its evolution, and thus (ii) model checking occurs in a considerably smaller transition system. The latter achieves *cone-of-influence* reduction on the state-transition structure submitted for model checking.

We consider systems that are formally modeled (static) graph-based structures along with their possible (rule-based) dynamics, yielding reactive behaviors. In particular, the specification is in terms of a complex static structure given in terms of a graph (which may be sourced from some domain model), and reactive dynamism is modeled by graph transformations. Interpretation of such specifications is costly; although state-of-the-art techniques and tools (e.g., SPIN [21], NuSMV [7]) employ cone-of-influence reduction for traditional reactive systems, the interpretation step in the expressive, graph-based reactive systems we consider involves explicitly constructing the state space, typically amounting to graph isomorphism on each step. Instead of slicing a state-transition structure – which in our case is not available and must be constructed – we work at the model specification level [6, 28].

Our contributions lie within a technical framework integrating fundamental techniques to scale-up automated analysis by building upon the cone-of-influence intuition – that often only fractions of a model are necessary for reasoning on specific behaviors. Thus automatic projections can support verification even in cases where otherwise it would not be feasible for the full model. Specifically:

- We concretely support separation of concerns and abstraction by advocating multiple views tailored in an automated way for analysis of specific requirements; those enjoy
- sound model synchronizations having two constituents: (i) correctness of model projections, where projected submodels are sufficient for checking a given requirement, and (ii) correctness of synchronizations, where models involved are kept consistent under the "projection" relation. Finally,
- automated reflection facilities make use of bidirectional transformations, whose application to slicing, multiple-view analysis and model checking was not investigated before.

[1] Model slicing generalizes to models the program slicing technique introduced by [48].

Our modeling approach is based on bigraphs and Bigraphical Reactive Systems (BRS [36]), a graph-based modeling formalism proposed by R. Milner able to encompass other formalisms such as process calculi and Petri nets. Our motivations for choosing this formalism are (i) its generality, as bigraphs have seen applications on systems ranging from cloud to cyber-physical, (ii) its well-defined semantics and hierarchical structure leading to elegant algorithmic treatment, and (iii) its natural relation to Graph Transformation Systems, rendering future practical adoption and tool support easier. Following a model-driven approach, other domain specific models can derive the BRS specification models we target. To provide concrete evidence of the proposed model-based approach, we demonstrate that the goal of making analysis scalable through model projections can be achieved in practice on a characteristic case study within cyber-physical systems where multiple-view analysis is paramount to the design process.

The rest of the paper is structured as follows. Sec. 2 provides necessary background, while Sec. 3 gives an overview of the proposed approach. Sec. 4 introduces a type-based approach to automatically generate requirement views, and Sec. 5 describes model synchronization. Sec. 6 provides an assessment over a case study. Related work is considered in Sec. 7, and Sec. 8 concludes the paper.

2 BACKGROUND

In this section, we start with a simple scenario serving as a running example throughout the paper. Subsequently, we succinctly illustrate how to model graph-based structure (Sec. 2.1) and dynamics (Sec. 2.2), i.e., possible ways in which the system may change over time through actions. Then, we show (Sec. 2.3) how the resulting reactive system can be interpreted as a state machine, upon which requirements may be verified by model checking.

Running Example. Consider the case where a resource constrained mobile robot patrols a spatial domain to locate intruders. The spatial domain is divided into areas upon which a local *edge* server is located, supporting robots to perform computationally-intensive tasks (e.g., image recognition, to visually detect intruders) without the latency that a cloud connection would incur. Computation offloading is performed on Virtual Machines (VMs), to which the edge server is the host. As the robot moves within areas, a VM may need to be migrated to the respective server located in another area. To this end, network gateways may connect edge servers, through which migration occurs. The ubiquitous system should fulfill two requirements: (RQ1) an edge server should not be idle, if there is another one connected to it hosting 3 VMs (*load balancing*), and (RQ2) if there exists an intruder in the system, the robot must keep surveying areas "one", "two" and "three" in sequence, one after the other (in a *sequenced patrolling [34]* pattern).

Notice that the example describes a simple reactive system. A model of the system can be constructed, which represents structure, entities and their relations within the scenario, as well as dynamic evolution due to actions. Subsequently, interpretation as a state-transition model and its verification in the form of model checking can be used to check if the requirements hold. However, interpretation and verification of large models may be impractical. Moreover, multiple stakeholders (e.g., robotic experts, deployment engineers) may have different interests in the system, so separation of concerns is highly desirable. Following such a principle implies

supporting analysis on each concern separately and abstracting away details that are irrelevant.

2.1 Modeling with Bigraphs

Modeling systems within software engineering is certainly a wide theme. Our formalism of choice is bigraphs [36], a graph-based process meta-calculus that is able to capture widely different systems, while featuring interesting properties. A bigraph consists of two graphs. A *place graph* is a forest, a set of trees defined over a set of nodes, while a *link graph* is a hypergraph over the same set of nodes and a set of edges each linking an arbitrary number of nodes. Connections of an edge with its nodes are called *ports*. Place and link graphs are orthogonal, and edges between nodes can cross locality boundaries. Nodes that appear in a bigraph are typed[2]. What follows is an informal presentation of bigraphs as used in the scope of this paper, recalling definitions in [36, 45].

$P.Q$	*Nesting* (P *contains* Q)	(1a)
$P \mid Q$	*Juxtaposition of nodes*	(1b)
$-_i$	*Site numbered* i	(1c)
K_w	*Node with type* K *having ports* w	(1d)
$W \parallel R$	*Juxtaposition of bigraphs*	(1e)

Bigraphs can be described through a rigorous graphical representation as well as equivalent algebraic expressions (Formulae 1a-1e). P, Q, and K are names that define a node's type. Nodes can be structured hierarchically; the containment relationship is expressed in Formula 1a and is graphically described by nesting. Bigraphs may be placed at the same hierarchical structure level, as shown in Formula 1b. Additionally, bigraphs can contain sites (Formula 1c) that can be used to denote placeholders, i.e., the presence of unspecified nodes, graphically represented as shaded boxes. Absence of a site signifies no unspecified nodes in that part of the hierarchy. Each node type ("control") can be associated with a number of named ports. In Formula 1d the node identified by type K has port names w; ports are graphically represented as black bullets. Bigraphs can be contained in roots that delimit different hierarchical structures, thus being juxtaposed. In Formula 1e W and R are different roots.

An instance of the example system is illustrated in Fig. 1, where entities such as robots and servers are within three areas. Notice how areas are linked to names identifying them, and links of servers to the network gateway traverse the hierarchies (i.e., the nesting of nodes). Sites denote unspecified nodes. Using the algebraic notation, the bigraph of Fig. 1 can be represented as in Formula 2.

$$\mathsf{Area}_1.(\mathsf{Robot} \mid \mathsf{Server}_{\mathsf{lnk}}.(\mathsf{VM}|\mathsf{VM})) \mid \mathsf{Network}_{\mathsf{lnk}} \quad (2)$$
$$\mid \mathsf{Area}_2.(\mathsf{Intruder} \mid \mathsf{Server}_{\mathsf{lnk}}.\mathsf{VM} \mid -_1) \mid \mathsf{Area}_3.(\mathsf{Intruder} \mid \mathsf{Server}_{\mathsf{lnk}} \mid -_0)$$

Formally, a bigraph arises from two superimposed relations. Let V_B be a set of nodes and K the set of types of nodes. Let $ctrl_B : V_B \to K$ be a typing function, called type map. A *place graph* is a tuple $B^P = (V_B, ctrl_B, prnt_B, K)$ where $prnt_B : V_B \to V_B$ is an acyclic parent mapping modeling nesting. A *link graph* is a tuple $B^L = (V_B, E_B, ctrl_B, link_B, K)$ where E_B is a finite set of edges and $link_B : E_B \to 2^{V_B}$ is a link mapping assigning each edge the set of nodes which are connected by that edge. Then, a bigraph B consists of B^P and B^L: $B = (V_B, E_B, ctrl_B, prnt_B, link_B, K)$. Note that for B of Formula 2, $K = \{\mathsf{Robot}, \mathsf{VM}, \mathsf{Area}\ldots\}$.

[2]Types are called *controls* in bigraphical terminology.

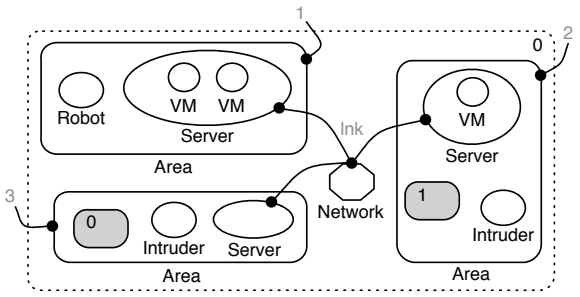


Figure 1: Bigraphical model of a ubiquitous system.

2.2 Modeling Dynamics with BRS

A Bigraphical Reactive System (BRS) [36] captures dynamic behavior. A BRS describes possible ways with which a bigraphical structure can evolve through application of transformation rules –called *reaction* rules– which selectively rewrite parts of a bigraph. Reaction rules have the general form of $\mathsf{R} \to \mathsf{R}'$, where the left-hand-side R (the *redex*) represents a pattern to be found in a bigraph, and a right-hand-side R' (the *reactum*) will replace a matched portion upon application of the reaction. R and R' are also bigraphs. Replacement of a subgraph matching a reaction rule redex with the subgraph defined by the reactum occurs in a fashion similar to graph rewriting [10], in a procedure called *bigraph matching* [36]. Rewriting procedures are computationally costly, as they are equivalent to graph isomorphism.

Utilizing reaction rules, dynamics of the example scenario can be modeled. The possible changes entail mobile robots changing areas (*move*), VMs migrating from connected servers (*migrate*), and the robot successfully capturing an intruder (*capture*), modeled as follows (a, b, c are variables ranging over named ports):

$$\text{(move)} \quad \mathsf{Area}_a.(\mathsf{Robot} \mid -_0) \mid \mathsf{Area}_b.(-_1) \to \mathsf{Area}_a.(-_0) \mid \mathsf{Area}_b.(\mathsf{Robot} \mid -_1)$$
$$\text{(migrate)} \quad \mathsf{Server}_{\mathsf{lnk}}.(\mathsf{VM} \mid -_0)) \parallel \mathsf{Network}_{\mathsf{lnk}} \parallel \mathsf{Server}_{\mathsf{lnk}}.(-_1))$$
$$\to \mathsf{Server}_{\mathsf{lnk}}.(-_0)) \parallel \mathsf{Network}_{\mathsf{lnk}} \parallel \mathsf{Server}_{\mathsf{lnk}}.(\mathsf{VM}_{\mathsf{lnk}} \mid -_1))$$
$$\text{(capture)} \quad \mathsf{Area}_a.(\mathsf{Robot} \mid \mathsf{Intruder} \mid -_0) \to \mathsf{Area}_a.(\mathsf{Robot} \mid -_0)$$

2.3 Analysis of Reactive Behaviors

Given an initial configuration specified by a bigraph and a set of reaction rules, new configurations may be generated by repeatedly applying reactions, and possible evolutions can be described by a state-transition structure [45]. This is a (doubly) *Labelled Transition System* [8] (dLTS) $\mathcal{L}$, defined as a tuple $\langle S, i, A, AP, \to, L \rangle$, where:

- S is a set of states describing configurations;
- $i \in S$ is the initial state;
- A is a set of transition labels;
- AP is a set of atomic propositions;
- $\to \subseteq S \times A \times S$ is a 3-adic relation of labelled transitions. If $p, q \in S$ and $\alpha \in A$, $(p, \alpha, q) \in \to$ is written as $p \xrightarrow{\alpha} q$.
- $L : S \to 2^{AP}$ is a function that labels each state with the set of propositions that are true in that state.

A BRS specification can be translated into an equivalent dLTS. Intuitively, given an initial configuration and a set of reaction rules, a dLTS can be generated by mapping bigraphical configurations onto states. The set of of propositions AP' that label a state ($p \in S$) can be systematically generated by declaratively encoding the corresponding bigraph configuration of the state. Transitions correspond to

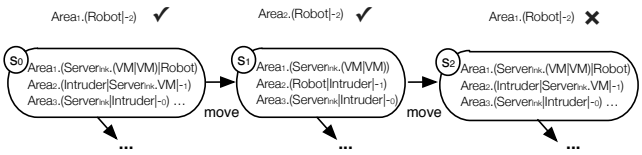


Figure 2: Evaluation of a property on a fragment of $\mathcal{L}$.

applications of reaction rules that lead to new bigraphical configurations and their labels record actions modeled by rules. An execution fragment ρ of $\mathcal{L}$ is a (possibly infinite) alternating sequence of states $s_i \in S$ and transition labels $\alpha_i \in A$, written as:

$$\rho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots s_{n-1} \xrightarrow{\alpha_n} s_n \dots$$

such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $i \geq 0$.

Behavior of the system over time (i.e., execution fragments ρ) can be reasoned upon with a temporal logic. We adopt Linear Temporal Logic [8] without the next operator (LTL_x [24]), defined as follows:

$$\phi ::= \text{true} \mid \alpha \mid \neg\phi \mid \phi \vee \phi \mid \phi \cup \phi.$$

Propositions of the logic are bigraphs, interpreted over states of $\mathcal{L}$. Such bigraphs are termed *parametric patterns*, since sites can be used to reason about unspecified nodes, and variables (e.g., a) can match port names – for instance, $\text{Area}_a.(-_0)$ would match all areas of the model of Fig. 1. The properties modeled by LTL_x express behavioral constraints and are interpreted over execution fragments of $\mathcal{L}$. Intuitively, the formula $\phi_1 \cup \phi_2$ expresses that ϕ_1 is true until ϕ_2 becomes true; we can derive additional ones such as $\diamond\phi = true \cup \phi$ ("eventually") and $\Box\phi = \neg \diamond \neg\phi$ ("always"). Then, requirements RQ1 and RQ2 of the running example can be formally specified by Formulae 4b and 4a:

$$RQ1: \ \Box \neg(\text{Server}_{\text{lnk}} \parallel \text{Network}_{\text{lnk}} \parallel \text{Server}_{\text{lnk}}.(\text{VM} \mid \text{VM} \mid \text{VM})) \quad (4a)$$

$$RQ2: \ \Box(\text{Intruder} \rightarrow \Box(\diamond(\text{Area}_1.(\text{Robot} \mid -_1) \wedge \wedge \diamond(\text{Area}_2.(\text{Robot} \mid -_2)) \wedge \diamond(\text{Area}_3.(\text{Robot} \mid -_3))))) \quad (4b)$$

LTL_x properties over bigraphical propositions (as patterns) as previously defined may be readily checked upon dLTS $\mathcal{L}$. For an execution fragment of $\mathcal{L}$ and an LTL_x property, evaluation of the property entails i) finding truth values of bigraphical propositions on states and subsequently ii) verifying the property's temporal component over the sequence of states in the fragment. To evaluate bigraphical patterns on states, matching [36] is used, while LTL_x properties are evaluated using established verification methods [8]. The restriction to LTL_x is a known strategy [14, 30] to enable verification optimization techniques without significantly affecting expressiveness of the language for practical considerations [3].

Fig. 2 shows an execution fragment of $\mathcal{L}$. Bigraphical patterns are evaluated in every state, while temporal properties are evaluated over the sequence; in states s_0 and s_1, patterns (shown above) $\text{Area}_1.(\text{Robot} \mid -_0)$ and $\text{Area}_2.(\text{Robot} \mid -_0)$ are respectively true (and the property holds true); in s_1, the robot moves back to Area 1, rendering the patrolling property RQ2 false.

3 VIEW-BASED REASONING FRAMEWORK

Given a reactive system specification comprising of an initial configuration and a set of rewriting rules, analysis in the form of model checking can be performed to check that possible behaviors do not violate the stated requirements. Traditional analysis (shown in the lower part of Fig. 3) would entail considering the entire model and set of rules, generating a state-transition structure by exploring application of all rules, and finally performing model checking for each property encoding a different requirement. The cornerstone of our approach, shown in Fig. 3, is that for each property, subsets of both the initial configuration and the set of rules are taken into account instead. This follows the spirit of slicing techniques as applicable to models (i.e., not programs), where the slicing criterion is a temporal logic formula (i.e., and not program variables).

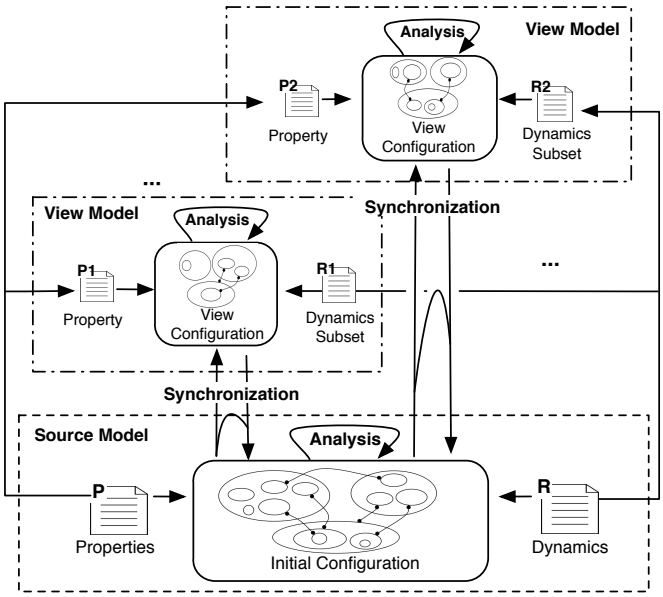


Figure 3: Framework for Multiple-View Reasoning.

The benefits of the proposed approach are twofold: (i) slicing is performed at the system specification level [4, 18], before the (computationally expensive) generation of the state-transition structure describing its evolution, and thus (ii) the size of the model submitted for model checking is reduced and tailored to formal analysis of each requirement. The result is known as achieving cone-of-influence reduction, which in this case has two dimensions. First, the *size of each state* can be reduced. Recall that each state is modeled by a set of propositions, and in our case propositions declaratively specify all details of any given bigraphical configuration. Our slicing approach automatically prunes all details that do not affect the requirement under consideration. Second, the *number of transitions* exiting each state (fan-out) can also be reduced.

The proposed approach not only supports effective mechanical verification but is also beneficial to designers in their reasoning about the current system. By projecting the full model into a more concise one which is equivalent with respect to satisfaction of a given requirement, it supports separation of concerns and abstraction. It helps focusing on each concern separately and automatically factoring away details that are irrelevant to a given requirement. For example, the robotic expert of the example system (Sec. 2) can focus her analysis on robotic mission aspects that affect the respective requirement, while the expert on load balancing can focus on others (in this case, VM migration). Such system concerns are captured in different analyzable properties as well as appropriate subsets of the model and dynamics (upper part of Fig. 3).

Finally, the technique is rooted into the theory of *bidirectional transformations*, which guarantees synchronization between models, consistency, and well-behavedness. As illustrated in Fig. 3, in case of change in one of the views, changes are automatically propagated both to the full model and other views, triggering analysis if required. For our example scenario, if e.g., the robotic expert changes the relevant submodel due to verification results of requirement RQ2, analysis can be triggered again on the submodel needed to check RQ1 as well.

4 PROPERTY-DRIVEN VIEW GENERATION

Let us suppose that a reactive system specification model is given by providing a bigraph describing an initial configuration (e.g., Fig. 1) and a set of reaction rules describing dynamics (as per Sec. 2.2). This section describes how a *view model* can be generated automatically, given a property. The approach is requirement-driven and generates a view model, which is a projection of the full model. The projected model comprises an initial configuration that is a subgraph of the original, as well as a subset of the original rules. The description is illustrated by referring to the running example. We first introduce an algorithm that generates a view model for a given property (Sec. 4.1). Subsequently, Sec. 4.2 outlines the proof of correctness based on previous results achieved within model slicing.

4.1 Views through Type-based Transformation

The algorithm for view model generation has as input a full model M – consisting of a bigraph C and a set R of reaction rules describing dynamics – as well as an LTL_x property P specifying a requirement. The output is a *projection* of the full model into a submodel comprising (i) a subgraph C' of C and (ii) a subset R' of R such that satisfaction of P on the full model can be proved equivalently by checking the satisfaction of P on the view model. We define a model transformation where the *source* model is the full model and the target model is the *view* model. The proposed transformation is embedded into an implementation that supports *bidirectional model transformations*, as discussed in Sec. 5.

A model that contains exactly what is relevant to check satisfaction of a requirement and is minimal (an *optimal* model) could be carefully hand-crafted in many cases. However, manual construction is non-trivial and prone to human error. Instead, we aim for an automatic technique to generate model views per requirement, that are provably equivalent with respect to analysis of the requirement on the source model. In the running example, intuitively one can observe that verification of requirement *RQ2* does not require considering VMs: including or excluding the migrate rule in the dynamic analysis does not affect satisfaction or violation of that requirement. The presented algorithm utilizes types to generate views, by exploiting the correspondence between types appearing in requirements, initial configuration, and dynamics. The algorithm does not guarantee generation of a minimal view model; more on this point is discussed in Sec. 8. Recall that in a bigraphical model, different kinds of entities are reflected in types (*controls*) of the bigraphical representation. A requirement may concern entities having only certain types; consider Formula 4b, which only predicates about Area and Robot and Intruder node types. Thus, our approach to view generation builds on the basic intuition of non-inclusion of node types that are not relevant to a requirement.

Algorithm 1 Computing Types and Rules for a View Model.

Input:
P – Bigraphical temporal property
R – Set of reaction rules
Output:
PropTypes – Set of types needed for P
R_P – Set of rules needed for P

```
1:  PropTypes ⇐ ∅ ; R_P ⇐ ∅
2:  for all p ∈ prop(P) do
3:      PropTypes ⇐ PropTypes ∪ k(p)
4:  end for
5:  repeat
6:      PropTypes' ⇐ PropTypes
7:      for all r ∈ R do
8:          if k(rhs(r)) ∩ PropTypes ≠ ∅ then
9:              PropTypes ⇐ PropTypes ∪ k(lhs(r))
10:             R_P ⇐ { r } ∪ R_P
11:         end if
12:     end for
13: until PropTypes' == PropTypes
```

Algorithm 1 computes the set of types *PropTypes* needed to evaluate a property P, given a set of reaction rules R. It also computes a subset R_P of R, which includes the reaction rules we need to consider to evaluate property P. The algorithm uses two help functions, k and prop. Function k maps a given bigraph into the set of types of its nodes. Recall that a temporal property has bigraphical patterns as propositions; function prop maps a given property into all bigraphs found as propositions in the property (i.e., discarding modal temporal operators). The algorithm iteratively computes the set of types needed for the evaluation of the property, by considering the left-hand-sides and right-hand-sides of reaction rules. The result is a subset of types that can be used to reduce the source model. In addition, the algorithm produces a subset of rules relevant to analysis of the property.

The algorithm starts by initializing *PropTypes* with all node types appearing in bigraphical propositions of the property (lines 2-3). For the property specifying RQ1, the algorithm will include types {Server, Network, VM}. Rules need to be also taken into account since sequences representing change of truth values of propositions might affect satisfaction or violation of bigraphical propositions within P. To this end, every rule r in the set of reaction rules R needs to be checked whether it is related to P. Specifically, while iterating through reaction rules (line 6), if the right-hand-side of a rule has types in common with *PropTypes* (line 9), the types appearing in the left-hand-side of the rule are also included in *PropTypes* (line 10). For example, for the rule *migrate*, k(rhs(migrate)) ∩ *PropTypes* = {Server, Network, VM} and k(lhs(migrate)) = {Server, Network, VM}; thus, no type will be additionally included to *PropTypes* as they have already been there. The algorithm continues to iterate until reaching a fixed point, i.e., the set *PropTypes* does not change (line 12).

To generate a view model $M_P =< C_P, R_P >$ from a model $M =< C, R >$, Algorithm 1 is used to produce R_P and the set of types *PropTypes* needed for evaluation of P . The latter is then used to generate a projected configuration C_P by pruning the configuration C as described in Algorithm 2. We take advantage of the fact that configurations are modeled as bigraphs, whose nodes are

Algorithm 2 Generating a View Model.

pruneNode : (**N, PropTypes**) → **N′**
N, N' are nodes of the form $Q.(NL)$
where NL is a list of nodes with the recursive form $N \mid NL$
1: $NL' \Leftarrow$ **pruneNodeList**(NL, $PropTypes$)
2: **if** $k(Q) \cap PropTypes \equiv \emptyset$ and NL' is $\emptyset$ **then**
3: $N' \Leftarrow null$
4: **else**
5: $N' \Leftarrow Q.(NL')$
6: **end if**

pruneNodeList : (**NL, PropTypes**) → **NL′**
1: **if** $NL \neq \emptyset$ and in the form of $N_{head} \mid NL_{tail}$ **then**
2: $N'_{head} \Leftarrow$ pruneNode(N_{head},$PropTypes$)
3: $NL' \Leftarrow N'_{head} \mid$ pruneNodeList(NL_{tail}, $PropTypes$)
4: **end if**

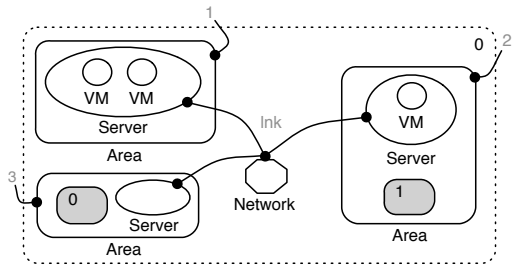
arranged hierarchically (the place graph, Sec. 2). For each hierarchical layer, there might be several nodes (e.g., different Areas) which are considered as a list. The objective of `pruneNode` is to prune a node, while `pruneNodeList` handles the list of nodes in order. In `pruneNode`, the function `pruneNodeList` is invoked first to handle nodes contained in that node (e.g., Robot and Server are contained in an Area). The node shall be deleted if the control of that node is not found in $PropTypes$ and none of the nodes contained in it does (line 2, i.e., NL' is null). Generation of the view configuration needs to ensure that: i) all nodes of the source configuration that have types in $PropTypes$ are included in the produced configuration, and ii) all their ascendants in the tree structure are also included, to ensure that structure is preserved. Sites are treated as having a special type, and are always included in the produced configuration – since they are placeholders for unspecified nodes. Fig. 4 presents the view models (initial configuration and set of rules) produced for requirements RQ1 and RQ2.

4.2 Correctness of Model Projections

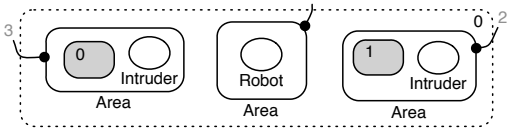
Projection of a reactive system source model into a view model for a given property P should yield a submodel equivalent to the full model as far as property P is concerned. This section provides the outline of a formal proof of this equivalence, leveraging theoretical results in model slicing achieved in the past [3, 24, 35].

A view model produced by the method described in Sec. 4.1 satisfies P if and only if the source model satisfies P as well. Since P is a temporal property, the equivalence has to be shown across execution fragments. The proof outline leverages insensitiveness to *stutter* within temporal logic; this means that a property can not distinguish between fragments that differ only by stuttering, i.e., finite repetition of similar states. Insensitiveness to stutter is a prerequisite for widely used techniques in software engineering such as partial order reduction [41] and model slicing [3]. Recall that property specification occurs within LTL_x, characterized by absence of the next-time LTL operator. The next operator is sensitive to finite stuttering, and a view projection essentially works by projecting a behavior and collapsing sequences of stuttered states. Any LTL_x formula describes a stutter-invariant property [42].

Given a model M consisting of a bigraph C describing an initial configuration, a set of reaction rules R describing dynamics and a property P, a view model $M_P = < C_P, R_P >$, where $R_P \subseteq R$, is



(a) $< C_{P1}, \{migrate\} >$.



(b) $< C_{P2}, \{move, capture\} >$.

Figure 4: View models generated for RQ1 (a) and RQ2 (b). Note the absence of Robot in C_{P1}, and of VMs in C_{P2}.

produced. Initially we introduce the concept of a *canonical form* of execution fragments, then we show that there is a *bisimulation relation* between the M's execution fragments and M_P's execution fragments. The proof outline consists of three steps. First, we show that any execution fragment affecting P obtained from the full model M has an equivalent canonical execution fragment where transition labels only belong to R_P. We call these fragments *canonical* because they are minimal with respect to P. In our context, equivalent signifies that property P either holds or does not hold for both execution fragments [3]. Second, we show that for each canonical execution fragment obtained from the full model M there is an execution fragment in the view model M_P which is P-equivalent. Third, we show that for each execution fragment obtained from the submodel M_P there exists a canonical form fragment obtained from $< C, R >$ which is equivalent.

Canonical fragment equivalence in the full model. Consider the following fragment obtained from the full model $M = < C, R >$, which captures application of rules $\alpha_i \in A$ from the initial state s_0:

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots s_{n-1} \xrightarrow{\alpha_n} s_n \ldots \tag{5}$$

The basic intuition behind the canonical form execution fragment is that if $\alpha_j \in R \setminus R_P$, α_j does not modify any part of the bigraph of state s_j that will affect the evaluation of the property. An execution fragment obtained from $< C, R >$ may contain labels corresponding to rules from both R_P and $R \setminus R_P$ intermixed. To obtain the corresponding canonical form fragment, any sub-sequences of transitions with labels in $R \setminus R_P$ can be dropped. For example, consider the case where in execution fragment (5), $a_j \in R_P$ for all $1 \leq j < i$ and $j \geq i+k$, and $a_j \in R \setminus R_P$ for all $i \leq j < i+k$. The corresponding canonical form fragment is:

$$s_0 \xrightarrow{\alpha_1 \ldots \alpha_{i-1}} s_i \xrightarrow{\alpha_{i+k} \ldots \alpha_n} s_n \ldots \tag{6}$$

Recall that rules in $R \setminus R_P$ do not affect satisfaction of P. For any execution fragment, one (and only one) canonical form exists. It can be easily shown that the canonical fragment is indeed an execution fragment of the full model.

Source-View fragment equivalence. As we observed, given any execution fragment of M, we can produce a canonical execution

fragment where all transitions are labeled with rules in R_P:

$$s_0 \xrightarrow{\alpha_1} s_1 \ldots \xrightarrow{\alpha_n} s_n \ldots \tag{7}$$

For the canonical execution fragment (7), by applying the procedure described in Sec. 4.1 on bigraphs that label states $s_0, s_1 \ldots s_n$ to eliminate from the configurations all nodes whose types do not affect property P, we obtain the following execution fragment:

$$s_{P_0} \xrightarrow{\alpha_1} s_{P_1} \ldots \xrightarrow{\alpha_n} s_{P_n} \ldots \tag{8}$$

where the atomic proposition labeling state s_{P_0}, is a bigraph C_P and for all $i \geq 1$, $\alpha_i \in R_P$ and the atomic proposition labeling state s_{P_i} is a bigraph obtained from the one labeling $s_{P_{i-1}}$ through the application of the rule α_i. Fragment (8) is an execution fragment of M_P and is equivalent to fragment (7). Thus, for each execution fragment obtained from the full model there is an execution fragment in the view model which is equivalent with respect to property P.

Canonical fragment equivalence in the view model. Consider any execution fragment obtained from $M_P = < C_P, R_P >$, for example fragment (8). If we apply the same sequence of transitions α_i for all $i \geq 1$ to an initial state labeled by C, we obtain a canonical execution fragment of M that is equivalent to fragment (8).

5 MODEL SYNCHRONIZATION

We have shown that a view model can be generated from a source model according to a specific requirement. However, a classic problem in this view-based approach is how to reflect changes from the view to the source model. If an update is made to the view model after verification (e.g., to counteract a requirement violation found), we should be able to reflect it to the source model, and from there to other view models that might also be affected. In this section, we show how to solve this problem by designing and implementing a bidirectional transformation (BX) for synchronizing the source and the view models to correctly propagate changes between them.

Without loss of generality, the following description only discusses how changes in view configurations may affect model synchronizations (and vice versa). Changes to dynamics can be handled similarly. Precisely, we assume the set *PropTypes* for a view not to change, and hence also the set of dynamic rules. Thus, whenever we use the term *model* in this section, it stands for only the static part (i.e., the bigraphical *configuration*).

5.1 A BX Algorithm

Bidirectional transformation (BX) [11, 17] is a useful mechanism for data synchronization, which will be used for synchronization of source and view models. A BX consists of a pair of transformations *get* and *put*. The *forward* transformation $get(s)$ is used to produce a target view v from a source s, while the *putback* transformation $put(s, v)$ is used to reflect updates on the view v to the source s. These two transformations should be *well-behaved* in the sense that they satisfy the following round-tripping laws:

$$\begin{array}{lcll} put(s, get(s)) & = & s & \textsc{GetPut} \\ get(put(s, v)) & = & v & \textsc{PutGet} \end{array}$$

The GetPut property requires that no change of the view shall be reflected as no change of the source, while the PutGet property requires all changes in the view to be completely reflected to the source so that the changed view can be computed again by applying the forward transformation to the updated source.

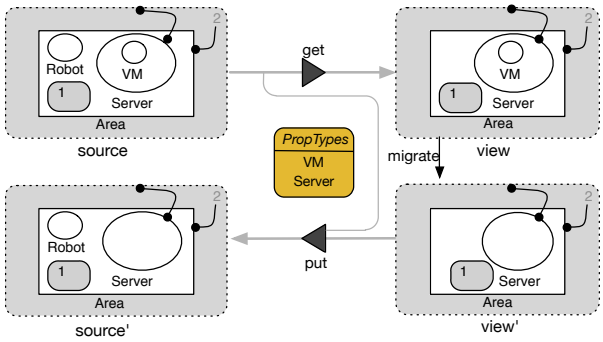


Figure 5: Type-based Bidirectional Model Transformation.

Algorithm 3 Updating a View Model.

updateNode : ($\mathbf{N_s}$, $\mathbf{N_v}$, **PropTypes**) $\rightarrow \mathbf{N'_s}$
N_s, N_v, N'_s are nodes of the form $Q.(NL)$.
1: $Q'_s \Leftarrow$ update attributes of Q_s with Q_v
2: $NL'_s \Leftarrow$ **updateNodeList**(NL_s, NL_v, *PropTypes*)

updateNodeList : ($\mathbf{NL_s}$, $\mathbf{NL_v}$, **PropTypes**) $\rightarrow \mathbf{NL'_s}$
NL_s, NL_v, $NL_s{}'$ are lists of nodes of the recursive form $N \mid NL$.
1: **if** both NL_s and NL_v are $\emptyset$ **then**
2: End Update
3: **else if** match $N_{s_{head}}$ with $N_{v_{head}}$ successfully **then**
4: $N_{s_{head}}{}' \Leftarrow$ **updateNode**($N_{s_{head}}$, $N_{v_{head}}$, *PropTypes*)
5: **else if** $k(N_{s_{head}}) \cap PropTypes \equiv \emptyset$ **then**
6: $N_{s_{head}}{}' \Leftarrow N_{s_{head}}$
7: **else if** $k(N_{s_{head}}) \cap PropTypes \neq \emptyset$ **then**
8: $N_{s_{head}}{}' \Leftarrow$ delete node (contained in) $N_{s_{head}}$ in *PropTypes*
9: **else if** NL_s is $\emptyset$ **then**
10: $NL_s{}' \Leftarrow NL_v$
11: **else**
12: **raise exception**(view NL_v is illegal)
13: **end if**
14: $NL_{s_{tail}}{}' \Leftarrow$ **updateNodeList**($NL_{s_{tail}}$, $NL_{v_{tail}}$, *PropTypes*)

To use BX for synchronization, we need to develop a pair of transformations between source and view models. The forward transformation *get* is straightforward, which is the view generation function (Sec. 4.1) when a set of types *PropTypes* is given:

$$get(s) = \texttt{pruneNode}(s, propTypes).$$

What is not so obvious, is how to define the corresponding *put* that can be paired with *get* to form a BX satisfying the GetPut and PutGet properties. Before defining *put*, let us consider a concrete example to see what this BX should be. As shown in Fig. 5 for the running example and the load balancing requirement $RQ1$, the *get* produces the view model with the types of VM and Server. When the *view* is changed to *view'* (VM is migrated), we hope to use *put* to propagate this change back to *source*, yielding the modified *source'* (VM moves away while the rest of the the model remains the same).

Since our *get* is a projection, its corresponding *put* should be an embedding of the view elements into the source model. To define *put*, we exploit the hierarchical tree structures of bigraphs echoing the previous view generation transformation strategy, and use the view to update the source layer by layer; *put* is defined by:

$$put(s, v) = \texttt{updateNode}(s, v, PropTypes),$$

where updateNode, informally defined by Algorithm 3, replaces the attributes (i.e., ports) of the source with the information in the view, and then invokes function updateNodeList to handle nodes contained in the source node. Function updateNodeList deals with the situation where both the source and view are a list of nodes. The simplest strategy for updating a list is position-based alignment, which matches the source and view elements by their positions in the lists. There are 6 cases: 1) both source and view are empty and we reach the end of execution; 2) the first node of the source can be matched with the first of the view (e.g., Server in the source can be matched with Server in the view), and updateNode is called to update this node; 3) controls in the first node (including all nodes contained in it) of the source are not found in *PropTypes* (i.e., no information shall be extracted to the view), in which case this element of the source will remain the same; 4) some controls in the first element belong to *PropTypes*, then this node and its containing ones with controls found in *PropTypes* are deleted; 5) the source is empty but the view is not (i.e., the source list has fewer elements); the new source nodes have to be created; 6) otherwise the view is not legal, and the computation fails to guarantee well-behavedness. Later, updateNodeList will be invoked recursively to manage the rest of the source and view lists. This *put* update strategy can be paired with the *get* defined in Sec. 4.1 to form a well-behaved BX, though the formal proof is omitted in this paper.

5.2 Implementing BX using BiGUL

The transformations *get* and *put* presented in Sec. 5.1 could be manually implemented. Although this solution provides the programmer with full control in two directions and can be realized using standard programming languages, maintenance effort is required to keep consistency between *get* and *put* if one of them is changed; even a small modification to one of the transformations would require redefinition of the other. Moreover, the pair of transformations should be proven to be correct and well-behaved.

To this end, we adopt BiGUL [25], a putback-based bidirectional programming language, where one is only required to implement the *put* transformation instead of both *get* and *put*. This is because *get* is uniquely determined by *put* based on well-behavedness [22]. In BiGUL, once a *put* transformation is given, the corresponding *get* is automatically derived. We will not dive into a detailed explanation of *put*, as its semantics are illustrated in Algorithm 3. Interestingly, from that *put* algorithm, BiGUL can automatically derive the *get*, with the same semantics as Algorithm 2. The interested reader is referred to [27] for the underlying mechanism of BiGUL, and to the online appendix for the implementation [1].

Use of Synchronization. Synchronization between views is useful for model maintenance purposes; recall the motivational example. Assume that the cloud expert decides to improve load balancing by dividing the available area into 4 areas, placing an edge server in each to increase system capacity. To this end, she changes the cloud view and proceeds to verify RQ1. The source and the robot view are updated automatically, and verification can be performed for RQ2 as well – notice that in this case, the robot view has changed as another area has appeared. Observe that RQ2 can be violated if the sequenced patrolling behaviour is not maintained.

6 EVALUATION

To provide tool support for our multiple-view analysis framework, we realized a prototypical view generation and transformation tool based on BiGUL [26], which is freely available [1]. Thereupon, we demonstrate the framework in practice over a characteristic case sourced from the domain of cyber-physical systems; experimental setup and results obtained are subsequently presented. We highlight a typical design scenario where multiple-view analysis and synchronization through bidirectional transformation is paramount to the design process and conclude with a discussion.

6.1 Multiple-Views in a Smart City Design

The prevalence of sensors, networks and devices has led to the emergence of smart urban environments. We consider two generalized cases which can be concretized for various scenarios; those concern different requirements but their analysis is based on the same model, obtained from a domain model of a city. Typically those may be analyzed separately by different stakeholders; however –as will later be illustrated– due to changes within a typical design workflow, a need of supporting synchronization among multiple views arises. We note that are not debating the underlying formalism here (e.g., for modeling CPS), as this is outside of the scope of this paper – our goal is investigating analysis scalability.

The first case concerns environmental monitoring with *Wireless Sensor Networks* (WSNs), comprised of small devices scattered in wide areas collecting measurement data. In order to gather data from low-powered sensors, one or more mobile *sinks* can be used. A sink is a gateway able to connect to a WSN device and also to larger networks like the Internet; a sink downloads data from a sensor when it is near it. This scenario entails verification of data collection; sinks move over the transportation network of the city, collecting data from sensors. The second case concerns *search and rescue*, a setting of emergency response where autonomous UAVs are dispatched to locate victims in the city [45]; UAVs move over buildings, looking for victims in need of assistance. We seek to verify that all victims in the city are eventually located by the swarm of UAVs, in all possible system behaviors.

6.2 Experiment Setup & Results

To utilize our approach in practice, the designer specifies the system model including its dynamics, as well as desired system properties. The views can then be produced automatically as described in Sec. 4.1. Implementation is available in accompanying material [1], along with models used and an experiment reproduction kit.

In the following, we illustrate the specification steps of the case study. Bigraphical models of the physical space of cities are automatically extracted [47] from randomly synthesized [5] domain models in CityGML[3]. The randomly synthesized models are to control for size while maintaining a canonical structure of real cities. Real city models can be used as well [47], but would distort results, the goal of this case study being to demonstrate scalability in a controllable setting. The bigraphical configuration contains various elements present in the city, such as *Blocks*, *Buildings*, *Roads* and *Crossroads*, linked accordingly to capture the topological structure inherent in the domain model [45]. Blocks may contain an arbitrary

[3] CityGML is a standard for urban modeling, with models existing for multiple cities.

Table 1: Experiment results on cyber-physical city models.

	Bigraph Size	# of Rules	dLTS St. (Trans.)	Interpret. Time	BX Time
city1–Source	307	24	3977 (23k)	46m	—
city1–UAV view	118	12	286 (1k)	2m	26ms
city1–WSN view	132	12	234 (1k)	13m	24ms
city2–Source	666	24	14046 (90k)	6h	—
city2–UAV view	293	12	1895 (8k)	16m	53ms
city2–WSN view	215	12	347 (2k)	3m	52ms
city3–Source	1198	24	—	>10h	—
city3–UAV view	601	12	4697 (30k)	1.5h	65ms
city3–WSN view	279	12	423 (2k)	4m	68ms

number of buildings; each is connected to the one next to it, and to a block's surrounding roads if it is located in the block boundary, while roads are linked to crossroads. Subsequently, sensors containing *DataToken*s, describing sensor deployment, and $Victim$s to be rescued are placed randomly; for the experiment setup, we consider 5 of each. Additionally, 4 *UAV* and *Sink* entities are also placed. Dynamics specification encodes (i) movement of data sinks across the transportation network in the city and data collection from sensors, and (ii) movement of UAVs and victim localization. The elements placed (UAVs, Sensors etc) are the same in all source models, in order to control for size: we seek to evaluate how increasing the problem size (i.e., city model size) affects scalability, while keeping dynamics stable. Since we are concerned with evaluating scalability – the reduction of transition system size and interpretation time over views – and not verification, we consider example LTL_x properties, aiming to (i) capture data collection by data sinks (e.g., $\diamond(\Box\neg\mathsf{DataToken})$) and (ii) ensure victim search and rescue by UAVs (e.g., $\Box(\mathsf{Victim} \rightarrow \diamond\neg\mathsf{Building}_?.(\mathsf{Victim} \mid -_0))$).

After specification, views are generated and interpreted as described in Sec. 2.3. For BRS interpretation, external tools can be used, e.g., [33, 44]. Thereupon, we report on experimental results. View generation leads to smaller models both in terms of size in nodes and ports (Column "Bigraph Size" in Tab. 1) as well as reduced dynamics sets, with both reduced state-transition size and more efficient interpretation times (Column "dLTS Size" and "Interpretation Time" in Tab. 1) due to reduced configuration sizes in states and less concurrency due to reduced dynamics. Progressively larger city models demonstrate this – note how the source model of $city3$ was unable to be interpreted. Experiments were performed on an Intel i5 3.1GHz; absolute values of interpretation time naturally depend on tools used – what matters in our setting is the improvement obtained by analyzing views, which drastically reduces space and time. Model checking on the transition systems finally produced can then be readily performed with typical tools (e.g., SPOT [13]).

Multiple-Views within the Smart City Workflow. After the analyses performed, we outline a scenario taking place in the smart city design process, in which model synchronization proves useful:

(1) A stakeholder in the system indicates that sensors must be placed additionally in city buildings for environmental monitoring. The city model is updated to reflect this.
(2) The change is propagated to the views corresponding to the WSN and UAV scenarios, and analysis of the respective requirements is triggered. The column "BX Time" in Table 1 records the time of projection for each new model. We note that view generation time is in the order of tens of milliseconds (source update time is similar, so omitted).
(3) Interpretation (and analysis) is performed yet again on the two views – times are quite similar to the ones in Table 1, so they are omitted. While analysis of the UAV requirement may be successful, the requirement corresponding to data collection is violated. Intuitively, since sensors are now also located in buildings, data collection from them cannot occur since sinks move only over the transportation network; a change in the system design is required.
(4) Since UAVs move between buildings, an upgrade to their hardware can render them equipped with data collection capabilities. The designer encodes an appropriate reaction rule, and interpretation (and analysis) on the views is triggered again; both requirements are finally satisfied.

Note that (i) in principle, every update in the views is supported and will be correctly propagated, and that (ii) stakeholders may analyze, debug or repair views, instead of the large source model.

6.3 Discussion & Limitations

As illustrated in the case study, the design process can be facilitated by reasoning on multiple views, and analysis scalability can be increased. Views can be produced via *get* and synchronization is achieved by *put* transformations. Note that the time used for generating views and synchronizing models can be neglected compared to the costly reactive system interpretation. However, as explained in the following, we note that (i) the view generation method used is conservative, and that (ii) future consideration of diverse case studies should assess applicability in other domains.

By performing analysis on the projected configurations and with reduced dynamics, view models benefit from the reduced concurrency and smaller size, resulting in smaller state-transition structures submitted for verification. However, in the scenarios presented, while respective requirements pertain a single model, they are to a certain degree disjoint as they capture different concerns. We acknowledge that in the case where models have increased overlap in types, there may be no increase in efficiency. This is due to the view generating Algorithm 1, which conservatively (but efficiently) only uses types to derive view models. A more advanced extension would refine the model further; we identify bigraph *matching* [36] as the key driver for further automatic view refinement.

To precisely discuss scalability, we must consider that the analysis technique we target is explicit-state model checking, which relies on an exhaustive exploration of the state space; this intrinsically requires restricting a *scope* to be feasible. Exhaustive state exploration is motivated by the *small scope hypothesis*, which states that a high proportion of bugs can be found by examining a system within a small scope [2, 23]. The approach presented allows to scale up the boundaries within which it can be performed with acceptable performance. The results obtained indicate that the technique is effective within a model and workflow of a CPS – stakeholders in such a scenario typically work with overlapping views of the same system (e.g., the physical space of the city). Practical experience with other domains would provide additional useful assessments and be an interesting future effort. Specifically, a systematic investigation would assess if meaningful properties and view models in typical use cases in other domains exhibit overlap in types.

Our multiple-view technique has been demonstrated as based on bigraphs and BRS, which although generic and widely applicable have not enjoyed e.g., practical applications. However, their well-defined semantics and hierarchical structure led to elegant algorithmic treatment (Sec. 4.1 and 5.1). We note that the technique could be readily applied for generating reduced views for other graph-based formalisms that utilize types, such as Graph Transformation Systems, with similar procedures and adoption of BX to support synchronization across models, guaranteeing consistency.

Finally and regarding the overall process, a typical design workflow as illustrated in Sec. 6.2 shows the benefit of view-based reasoning through synchronizations automatically updating models when changes occur. Moreover, the performance of transformations (rightmost column of Table 1) renders the procedure applicable to an online setting. To this end, we believe that integration to a toolchain supporting design and analysis as illustrated in this paper has merit and can enable practical uses. User-facing issues pertaining to integration to some workflow or pipeline, such as conflicts handling (where e.g., two designers concurrently edit two views) or merging should be tackled as well, along with general usability aspects.

7 RELATED WORK

Our technique is founded on the idea that multiple views are needed to reason about a system for different requirements. Accordingly, we classify related work into multiple-view reasoning, model slicing, and related approaches of synchronization with BX.

Views as first-class citizens have been introduced by Nuseibeh et al. in their work on requirements of composite systems [16, 38], where requirements are often elicited from multiple sources expressing multiple and partial viewpoints. A follow-up proposes interaction and integration of different viewpoints contributing to resulting requirements specifications [39]. Subsequent work involved inconsistency handling between such viewpoints by supplying logical rules [15]. The importance of a rich model providing structure and integrating complementary views capturing different system facets was highlighted in [46]. Multi-view reasoning was further adopted in architectures with multiple potentially conflicting concerns for quality requirements within mission-critical systems [12]. Our reasoning framework is similarly based on multiple views; our goal however of view generation is analysis of different system facets according to a requirements specification.

Slicing has been proposed as a program analysis technique [48], to extract the parts of a program that affect (or are affected by) execution of a given statement. Slicing has also been successfully applied at the model level [3]; in proposition-based slicing [19] temporal logic formulae are used to reduce the size of a transition system for model checking. Wang et al [24] use slicing to reduce the state space of UML statecharts when model checking LTL_x. The slicing criterion typically consists of elements from a property ϕ, such as the set of states and transitions in ϕ, or the set of variables in ϕ (e.g., for programs). The slice produced must preserve the behaviour of those parts of the model that affect the truth value of ϕ [3]. The typical strategy to achieve this is a recursive application of an operation until a fixed point is reached - which is similar to the technique we employ upon a BRS specification. Our technique traverses bigraphical structures until a fixed point emerges, however we apply this on the system specification using types from bigraphical propositions within LTL_x properties. We note that our domain (graph-based reactive systems), problem domain (multiple-view analysis) and use of BX (to maintain consistency between slices) are to the best of our knowledge novel.

Bidirectional model transformations are a popular mechanism for maintaining consistency of at least two related sources of information. An approach that defines a consistency relation between two models is the QVT Relations language in the OMG QVT standard, supported by a tool complying to checking semantics [32]. A Triple Graph Grammar [43] can be used to conclude consistency, particularly between graph-like structures, as well as to find a partial correspondence model combined with linear optimization techniques to detect maximum consistency portions [31]. However, it is time-consuming and non-trivial to manually maintain round-tripping laws. Other approaches consider a standard forward-direction with automatically derived backward transformations, such as the Atlas language [49] or via graph querying [20]. However, the forward transformation may not be injective and its ambiguity of various corresponding put-directions is what makes bidirectional programming challenging in practice. Recently, putback-based approaches [22] have been proposed to only allow writing putback transformations. By contrast, a *put* transformation could uniquely determine *get* by well-behavedness, and the putback-based program guarantees that the *get* behaviours are unambiguously specified. BiYacc [50] and BiFlux [40] are typical examples. BiGUL is a formally verified language which serves as a foundation for higher-level putback-based languages [25, 26]. We adopt BiGUL in order to have full control over the consistency restoration behaviors.

8 CONCLUSION AND FUTURE WORK

Early requirements validation has been recognized as a fundamental software engineering principle. Within reactive systems, this can be achieved by providing formal high-level system and requirements specifications, their interpretation and mechanical verification that the system model formally satisfies the requirements. Fundamental problems however, are scalability and usability – this paper offers a novel approach that addresses exactly these problems. Regarding usability, the proposed approach supports separation of concerns, and thus the ability of different stakeholders to only focus on views that pertain to their interest. To support coherence of the entire system, a contribution of this work is the ability to keep views and specification synchronized, achieved through bidirectional transformations. Scalability is achieved by analyzing models tailored to specific requirements, instead of the complete system.

In future work, we plan to improve the way submodels are projected for a given property. The type-based approach we presented is conservative and can produce non-optimal submodels, although submodels are produced very efficiently. Instances of nodes instead of types might be taken into account to further reduce submodel size. The technique developed has been exploited in the case of a specific formalism based on bigraphs. The idea of view-based requirements validation, however, is more general and could be applied to generating reduced model views for other formalisms that utilize types. such as general Graph Transformation Systems. Likewise, bidirectional transformation techniques may be adopted in these other cases to support synchronization across models, guaranteeing consistency across system design and development.

ACKNOWLEDGEMENTS

Research partially supported by Austrian Science Fund (FWF) project M 2778-N "EDENSPACE" and the National Natural Science Foundation of China under Grant Nos. 61620106007 and 61751210.

REFERENCES

[1] 2020. Reproduction kit, models, and accompanying implementation. http://dsg.tuwien.ac.at/staff/ctsigkanos/ase20.
[2] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2003. Evaluating the small scope hypothesis. In *In Popl*, Vol. 2. Citeseer.
[3] Kelly Androutsopoulos, David Clark, Mark Harman, Jens Krinke, and Laurence Tratt. 2013. State-based model slicing: A survey. *ACM Computing Surveys (CSUR)* 45, 4 (2013), 53.
[4] Ramesh Bharadwaj and Constance L. Heitmeyer. 1999. Model Checking Complete Requirements Specifications Using Abstraction. *Autom. Softw. Eng.* 6, 1 (1999), 37–68.
[5] Filip Biljecki, Hugo Ledoux, and Jantien Stoter. 2016. Generation of multi-LOD 3D city models in CityGML with the procedural modelling engine Random3Dcity. *ISPRS Ann. Photogramm. Remote Sens. Spatial Inf. Sci.* (2016), 51–59.
[6] Ingo Brückner and Heike Wehrheim. 2005. Slicing Object-Z specifications for verification. In *International Conference of B and Z Users*. Springer, 414–433.
[7] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. 2000. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 410–425.
[8] E. M. Clarke, O. Grumberg, and D. A. Peled. 1999. *Model Checking*. MIT press.
[9] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Pasareanu, Hongjun Zheng, et al. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. IEEE, 439–448.
[10] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. 1997. Algebraic Approaches to Graph Transformation-Part I: Basic Concepts and Double Pushout Approach.. In *Handbook of Graph Grammars*. 163–246.
[11] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations, 2nd Intl. Conf, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings*. 260–283.
[12] Kadir Alpaslan Demir. 2015. Multi-View Software Architecture Design: Case Study of a Mission-Critical Defense System. *Computer and Information Science* 8, 4 (2015), 12–31.
[13] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proc. of the 14th Intl. Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, Vol. 9938. Springer, 122–129.
[14] Kousha Etessami. 1999. Stutter-invariant languages, ω-automata, and temporal logic. In *Intl. Conference on Computer Aided Verification*. Springer, 236–248.
[15] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. 1993. Inconsistency Handling in Multi-Perspective Specifications. In *Software Engineering - ESEC '93, 4th European Software Engineering Conference, Garmisch-Partenkirchen, Germany, September 13-17, 1993, Proceedings*. 84–99.
[16] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L. Finkelstein, and Michael Goedicke. 1992. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering* 2, 1 (1992), 31–57.
[17] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17.
[18] Vinod Ganapathy and S. Ramesh. 2002. Slicing Synchronous Reactive Programs. In *Electronic Notes in Theoretical Computer Science, 65(5). 1st Workshop on Synchronous Languages, Applications, and Programming*. Elsevier, Grenoble, France.
[19] John Hatcliff, Matthew B Dwyer, and Hongjun Zheng. 2000. Slicing software for model construction. *Higher-order and symbolic computation* 13, 4 (2000), 315–353.
[20] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. 2009. A compositional approach to bidirectional model transformation. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*. 235–238.
[21] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.
[22] Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. 2014. Validity Checking of Putback Transformations in Bidirectional Programming. In *FM 2014: Formal Methods - 19th Intl. Symposium, Singapore, May 12-16, 2014. Proc.* 1–15.
[23] Daniel Jackson and Craig Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Trans. Software Eng.* 22, 7 (1996), 484–495.
[24] Wang Ji, Dong Wei, and Qi Zhi-Chang. 2002. Slicing hierarchical automata for model checking UML statecharts. In *International Conference on Formal Engineering Methods*. Springer, 435–446.
[25] Hsiang-Shang Ko and Zhenjiang Hu. 2018. An axiomatic basis for bidirectional programming. *PACMPL* 2, POPL (2018), 41:1–41:29.
[26] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: a formally verified core language for putback-based bidirectional programming. In *Proc. of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 61–72.
[27] Hsiang-Shang Ko. 2013. BiGUL: The Bidirectional Generic Update Language. https://bitbucket.org/prl_tokyo/bigul.
[28] Sébastien Labbé and Jean-Pierre Gallois. 2008. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing* 20, 6 (2008), 563–595.
[29] Simon S Lam and A Udaya Shankar. 1984. Protocol verification via projections. *IEEE transactions on software engineering* 4 (1984), 325–342.
[30] Leslie Lamport. 1983. What good is temporal logic?. In *IFIP congress*, Vol. 83. 657–668.
[31] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. 2017. Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques. In *Fundamental Approaches to Software Engineering - 20th International Conference, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 191–207.
[32] Nuno Macedo and Alcino Cunha. 2013. Implementing QVT-R Bidirectional Model Transformations Using Alloy. In *Fundamental Approaches to Software Engineering - 16th Intl. Conference, FASE 2013, Rome, Italy, March 16-24, 2013. Proc.* 297–311.
[33] A. Mansutti, M. Miculan, and M. Peressotti. [n.d.]. Multi-agent Systems Design and Prototyping with Bigraphical Reactive Systems. In *14th IFIP WG 6.1 Intl. Conference, DAIS 2014, Berlin, Germany, June 3-5, 2014*.
[34] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger. 2019. Specification Patterns for Robotic Missions. *IEEE Transactions on Software Engineering* (2019), 1–1.
[35] Lynette I Millett and Tim Teitelbaum. 2000. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 343–349.
[36] Robin Milner. 2009. *The Space and Motion of Communicating Agents*. Cambridge University Press.
[37] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelsteiin. 2003. ViewPoints: meaningful relationships are difficult!. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 676–681.
[38] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. 1993. Expressing the Relationships Between Multiple Views in Requirements Specification. In *Proc. of the 15th Intl. Conf. on Software Engineering, USA, May 17-21, 1993*. 187–196.
[39] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. 1994. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Trans. Software Eng.* 20, 10 (1994), 760–773.
[40] Hugo Pacheco, Tao Zan, and Zhenjiang Hu. 2014. BiFluX: A Bidirectional Functional Update Language for XML. In *Proc. of the 16th International Symposium on Principles and Practice of Declarative Programming, UK, Sept. 8-10, 2014*. 147–158.
[41] Doron Peled. 1994. Combining partial order reductions with on-the-fly model-checking. In *International Conference on Computer Aided Verification*. Springer, 377–390.
[42] Doron Peled and Thomas Wilke. 1997. Stutter-invariant temporal properties are expressible without the next-time operator. *Inform. Process. Lett.* 63, 5 (1997), 243–246.
[43] Andy Schürr. 1994. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*. 151–163.
[44] Michele Sevegnani and Muffy Calder. 2015. Bigraphs with Sharing. *Theor. Comput. Sci.* 577 (2015), 43–73.
[45] Christos Tsigkanos, Timo Kehrer, and Carlo Ghezzi. 2017. Modeling and verification of evolving cyber-physical spaces. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, 2017*. 38–48.
[46] Axel van Lamsweerde. 2009. Building Multi-View System Models for Requirements Engineering. In *RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009*. 368–369.
[47] Ennio Visconti, Christos Tsigkanos, Zhenjiang Hu, and Carlo Ghezzi. 2019. Model-Driven Design of City Spaces via Bidirectional Transformations. In *Proceedings of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, September 15-20, 2019*. ACM.
[48] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.
[49] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. 2007. Towards automatic model synchronization from model transformations. In *22nd IEEE/ACM International Conference on Automated Software Engineering November 5-9, 2007, Atlanta, Georgia, USA*. 164–173.
[50] Zirun Zhu, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2015. BiYacc: Roll Your Parser and Reflective Printer into One. In *Proc. of the 4th Intl. Workshop on Bidirectional Transformations L'Aquila, Italy, 2015*. 43–50.