Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

TRANSLATING PROCESS INTERACTION WORLD VIEW MODELS TO DEVS: GPSS TO (PYTHON(P))DEVS

Randy Paredis
Simon Van Mierlo
Hans Vangheluwe

Department of Computer Science
University of Antwerp – Flanders Make
Middelheimlaan 1
Antwerp, BELGIUM

ABSTRACT

Discrete-event modelling and simulation languages can be classified based on their world view: event scheduling, activity scanning, or process interaction. To study the semantics of these languages one may investigate the relationship between them, and in particular translate models between languages in different world views. A translation approach also lets one re-use all the simulation tooling available for the target language. We describe a translation of the classic process interaction language GPSS developed by Gordon in the early 1960s onto DEVS, a modular discrete-event modelling and simulation language with precise semantics developed by Zeigler in the late 1970s. We specify and implement a translation that produces, for each GPSS model, a behaviourally equivalent DEVS model. As GPSS has no formal semantics, there is no proof of equivalence. Rather, we describe the structure of the translation, starting from Gordon's informal description, centered around the main data structures called chains and the scanning algorithm. We build a working prototype for a representative subset of GPSS blocks found in most tools implementing the language. Finally, we exhaustively test the translation by comparing simulation results of the generated DEVS model with a those obtained by the GPSS World simulator. GPSS World is a popular GPSS variant. We also demonstrate our approach on a small but representative example from the manufacturing domain.

1 INTRODUCTION

Modeling and Simulation (M&S) allows system engineers to tackle the inherent complexity of *designing* and *deploying* complex, software-intensive, cyber-physical systems, as well as to *analyze* existing systems. Multi-Paradigm Modeling (MPM) (Mosterman and Vangheluwe 2004) advocates explicit modeling all relevant aspects of the system using the most appropriate modeling language(s), at the most appropriate level(s) of abstraction. A requisite for these approaches is that *modelling languages*' syntax (the allowable constructs, as well as their textual/visual representation) and semantics (to unambiguously evaluate the models) are unambiguously specified. Modelling Language Engineering models language explicitly.

As a plethora of modeling languages exist and are used today, it becomes important to classify them according to their characteristics, so we can reason about them. (Overstreet and Nance 2004) provide a characterization of the three discrete-event "world views" that were previously introduced in (Zeigler 1976). In particular, they state that the three world views attempt to capture different types of locality: *event scheduling* provides locality of *time*, *activity scanning* provides locality of *state*, and *process interaction* provides locality of *object*. The authors describe the relation between the world views, and examine whether they can be transformed into each other. They conclude that these transformations are complicated since each world view makes certain implicit assumptions which need to be explicitly represented in the other world view(s).

In this paper, we study GPSS (Gordon 1978; Schriber 1974) as a representative example of the *process interaction* world view, and describe a translation onto *DEVS (Zeigler, Praehofer, and Kim 2000)*, a representative example of the *event scheduling* world view. Both languages have a long history over 45 years ago. In GPSS, the life-cycle of structured *entities* called *transactions* is modelled as a network of *blocks*. During the life of a transaction, it passes through blocks which may change those entities' attribute values and may keep them from moving. A transaction may be held in a block for a duration of time as it competes for limited resources, thus forming queues. Automatic gathering of statistics is also supported. DEVS is a general-purpose discrete-event modeling language, which has been described as a "assembly language" onto which other (discrete-event) languages can be mapped (Vangheluwe 2000a). Its main building blocks are *atomic models* that can be connected and that can exchange *events*. Models can be hierarchically composed into *coupled models*.

Structure Section 2 briefly introduces modelling language engineering, GPSS and DEVS. Section 3 describes GPSS to DEVS translation. Section 4 describes the translation of a GPSS model of a manufacturing system, a representative use case. Section 5 describes related work and Section 6 concludes the paper.

2 BACKGROUND

In this section, we explain the background necessary to understand the remainder of the paper. We start by introducing modeling language engineering, a framework that supports the design of modeling and simulation languages. Next, we explain the two formalisms used in the paper: GPSS and DEVS.

2.1 Modeling Language Engineering

Model-Driven Engineering (MDE) tries to bridge the gap between an engineer's knowledge of *what* a system should do and *how* that behavior is implemented. To achieve this, it advocates the use of *models* (abstractions) that specify aspects of the system's structure and behavior, created using *modelling languages*. Following *modeling language engineering* (Kleppe 2007) terminology, a modeling language is fully defined by: (1) its *abstract syntax*, defining the language constructs and their allowed combinations (typically captured in a metamodel); (2) its *concrete syntax*, specifying the textual/visual representation of the different constructs; and (3) its *semantics*, defining the meaning of models created in the language (Harel and Rumpe 2004).

For example, "`1 + 2`" and "`(+ 1 2)`" are both strings of characters. They act as textual concrete syntax for the abstract syntax: a tree structure with + as root and `1` and `2` leaves. The semantics, or "meaning", of the strings "`1 + 2`" and "`(+ 1 2)`" is the natural number 3. We distinguish two main categories of semantic definitions: (1) *operational semantics*, where the semantic mapping effectively executes, or simulates, the model, usually producing a trace; and (2) *translational semantics*, where the semantic mapping translates the model from one formalism to another. The target formalism has semantics (again, either translational or operational). The meaning of the original model, with respect to properties of interest, is thus given by the meaning of its translated version. In this document, we call a modelling language with well-defined semantics a *formalism*. When a formalism is fully specified, metamodeling environments can generate an interactive modeling and simulation environment from it. These techniques have been applied to synthesize an interactive visual modelling environment for GPSS in the past. In (de Lara and Vangheluwe 2002b), this was achieved with the AToM^3 metamodeling tool (de Lara and Vangheluwe 2002a). We will use this environment throughout the paper to model the GPSS examples that show our approach and to model and execute a translational semantics of GPSS by specifying a rule-based model transformation to DEVS.

If the semantics of a formalism are unambiguously defined, any implementation of that formalism (in a specific tool) needs to adhere to this specification; this means that, in theory, a model in the language simulated by different tools should produce identical simulation traces and satisfy the same set of properties. This is not always the case in practice, however. Problems may arise at different points: either (1) the specification of semantics leaves "holes" and/or (2) the different implementations of the formalism do not produce the same results for the same model. Such "holes" in the semantic definition might include underspecification with respect to scheduling, random number generation (distributions), etc. In this paper, we explain that the GPSS semantics are underspecified and define the semantics formally by mapping it onto DEVS, whose semantics are well-defined. To solve the ambiguity in the specification of the GPSS semantics, we make pragmatic choices based on existing implementations. The goal of the semantics presented in this paper is to be *equivalent* to the (precise) operational semantics of GPSS.

2.2 GPSS

GPSS (Gordon 1978) provides abstractions to model systems where entities compete for limited resources. The basic concepts are messages (*transactions*) traveling from a source (GENERATE block) to a sink (TERMINATE block) through a chain of connected blocks that perform modification operations. Its textual syntax follows a punch card format, while its graphical syntax consists of a network of blocks connected by arrows. There is no a strict, unambiguous, mathematical description. The semantics of all building blocks comes from a human-understandable meaning that was given to all blocks.

In GPSS, the notion of time is user-defined and can be anything that is required for the models to be simulated. When global information about the model is required, *standard numerical attributes* (SNAs) are used to obtain a snapshot value of an attribute of the system at that point in time. In this section we give a broad overview of GPSS and discuss a limited set of blocks (i.e. only the ones mentioned in this paper); the interested reader is referred to (Gordon 1978) for a complete discussion of GPSS.

Transactions in GPSS move through the modeled system. Each transaction stores information and allows state changes on this information. During execution, at most one transaction can be active at any given time, and it is the task of the simulator to update its state according to the specification in the model. A transaction stores a number of parameters, including a priority (PR), and the time at which it was spawned (M1). These parameters can be set upon creation, or later on, using the ASSIGN or MARK blocks.

Chains allow GPSS to coordinate the transactions and events that are being sent. Each transaction must either be on the current events chain (CEC), which contains all transactions that are scheduled for the current simulation time; the future events chain (FEC), which contains all transactions that are due to move at some future time; the interrupt chain, which contains all transactions that have seized a facility, but were interrupted during execution; a user chain (see later); or the match chain (out of scope).

The Scanning Algorithm is the heart of a GPSS simulation. This algorithm identifies which transaction is allowed to move when and how. It moves transactions over the different chains, keeping track of the overall system state and can be summarized as follows. All transactions in the system are sorted on the CEC by decreasing priority. One by one, they will travel as far as possible (within the current clock time) through a sequence of blocks. When a transaction is blocked, it is copied to a delay chain and the next unblocked transaction on the CEC is selected. A transaction that is scheduled to move at some point in the future (due to a GENERATE or an ADVANCE block) is placed on the FEC. When all transactions have been moved, the scan is restarted as long as there are transactions that can continue to the next block. If that's not possible anymore, the algorithm waits until the first transaction leaves the FEC before rescanning.

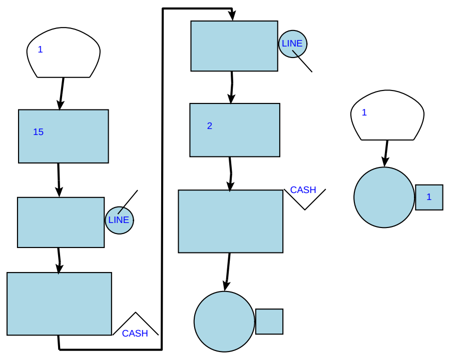
Resources can be accessed by a transaction, entering a *critical section*. If a transaction is denied access to a resource, it will be blocked and it will continue waiting in the block that requested access to that resource, except when using a GATE block that points to an alternative location. Whenever a resource is freed, all currently blocked transactions can retry to gain access to the resource. (Gordon 1978) states that this event triggers an immediate rescan, yet, after some experimentation, GPSS World (the implementation we used to simulate our models) appears to move the first item on the CEC to the next block before this rescan is fired. In the remainder of the paper, we will employ the GPSS World semantics. There are three types of resources: *facilities* (single access), *storages* (multiple access), and *logic switches* (boolean states).

Time can only be advanced by two blocks (by adding transactions to the FEC): GENERATE and ADVANCE. GENERATE blocks create transactions at a future point in time, while ADVANCE blocks delay each incoming transaction until some future point. Based upon a *random number generator* (RNG), the inter-arrival time between the messages in a *GENERATE* block, as well as the delay time for all incoming transactions in the *ADVANCE* are determined. The default distribution from which a value is chosen is $A \pm B$, with A a central point and B (possibly zero) the spread around it (Gordon 1978). Other distributions can be obtained using the FUNCTION statement that should identify the inverse cumulative distribution function. In that case, the RNG from GPSS generates a random value i.i.d. $U(0,1)$.

Statistics Gathering is natively supported by GPSS. The QUEUE, DEPART, MARK and TABULATE blocks can be used to monitor certain aspects of a model. Because of the lack of access to a resource, transactions may be blocked, implicitly forming a queue. To make this explicit, all transactions that find themselves between a QUEUE block and a DEPART block are monitored. To avoid confusion, we'll call this group of transactions a "queue", but note that this is not equal to the implicit queue that is formed due to blocking. The queue keeps track of the current and maximal queue length, the average transit time, the total number of entries and the number of entries that spent no time in the queue.

Other statistics, such as the rate of transactions traveling through a certain point and the total transit time can be recorded in tables, using the TABLE statement and the TABULATE block. The MARK block can be used to initialize a value to measure, while tables store gathered values in a set of buckets. The exact values will be lost, but the mean, standard deviation and counts for each bucket can be used for statistical analysis. Depending on the size of the buckets and the number of entries, the distribution of the statistic can be estimated as well.

Basic Example To demonstrate the use of GPSS, we model a store with a single cash registry. Each minute, a person enters the store; each person takes 15 minutes to buy their groceries. The checkout takes another two minutes. Figure 1 shows a possible solution to model this example system. The comments in Figure 1b provide an explanation for each line of the textual code, which each correspond to a block in the visual notation.



(a) Visual Notation in AToM^3.

```
* Simple Store Example

GENERATE    1        ; People entering the store.
ADVANCE     15       ; Shopping takes 15 minutes.
QUEUE       LINE     ; Wait in line before the registry.
SEIZE       CASH     ; There is but one cash registry.
DEPART      LINE     ; The person is not in the queue anymore!
ADVANCE     2        ; Checkout takes 2 minutes.
RELEASE     CASH     ; No need to keep the store clerk busy.
TERMINATE            ; People exit the store.


GENERATE    1        ; Make sure the simulation ends after a
TERMINATE   1        ;  certain time period.
```

(b) Textual Notation.

Figure 1: Basic GPSS Example.

2.3 DEVS

DEVS (Zeigler, Praehofer, and Kim 2000) is used to model the behaviour of discrete event systems. The basic building blocks of a DEVS model are *atomic DEVS* models, which are structures $< X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta >$, where the *input set X* denotes the set of admissible input events of the model. The *output set Y* denotes the set of admissible output events of the model. The input and output set of events define a set of input and output *ports*. These ports are used by a model to receive input events on or send output events to. The *state set S* is the set of sequential states of the model. The *internal transition function* $\delta_{int} : S \to S$ defines the next sequential state, depending on the current state. The *output function* $\lambda : S \to Y$ defines the output to be raised for a given sequential state, upon triggering the internal transition function. The *external transition function* $\delta_{ext} : Q \times X \to S$ with $Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ gets called whenever an *external input* ($\in X$) is received. The *time advance function* $ta : S \to \mathbb{R}^+_{0,+\infty}$ defines the duration the system remains in the current state before triggering its *internal transition function*.

A network of atomic DEVS models is defined by the structure $< X_\Delta, Y_\Delta, D, M_i, I_i, Z_{i,j}, select >$ where Δ is the DEVS network. The *input set X* and output set Y denote the sets of admissible input/output events of the network, similar to atomic DEVS models. D is a set of component references. M_i is the DEVS atomic model of component i, for all $i \in D$. I_i is the set of influencees of component i, for all $i \in D \cup \{\Delta\}$ (with $i \notin I_i$). $Z_{i,j}$ is the transfer function, for all $i \in D \cup \{\Delta\}$, and for all $j \in I_i$, where $Z_{\Delta,j}$: $X_\Delta \to X_j$; $Z_{i,\Delta}$: $Y_i \to Y_\Delta$; and $Z_{i,j}$: $Y_i \to X_j$. $select$: $2^D \to D$ is the select function. DEVS is closed under coupling; coupled models can be nested to arbitrary depth.

An abstract simulator for DEVS is described in (Muzy and Nutaro 2005). Tools such as PythonPDEVS (Van Tendeloo and Vangheluwe 2016) and adevs (Nutaro 2015), implement these semantics. A simulation step in the algorithm for the classic version of DEVS, as implemented by these tools, can be summarized as follows: (1) compute the set of atomic DEVS models whose internal transitions are scheduled to fire (imminent components); (2) select one imminent component with the *select* tie-breaking function; (3) execute the imminent component's output function, generating an output event; (4) route events from sending components to receiving components; (5) determine the type of transition to execute for each atomic DEVS model, depending on it being imminent or receiving input; (6) execute, in parallel, all enabled internal and external transition functions; (7) compute, for each atomic DEVS model, the time of its next internal transition.

3 TRANSLATING GPSS TO DEVS

In this section, we explain the main contribution of the paper: a translation procedure that maps GPSS models onto equivalent DEVS models. We start by explaining the choices made for implementing this translation, and then explain for each category of GPSS entities how they are translated to DEVS.

General Principles When translating GPSS to DEVS, multiple options for the translation are possible. A first option is to translate the complete GPSS model onto a single atomic DEVS model that implements the behavior of the GPSS simulator. The advantage of this approach is its high performance, since no communication overhead is introduced, while still allowing the model to be approached as a DEVS model. The disadvantage of this approach is that debugging the translated model is difficult, since any connection to the original model is lost. An alternative to this approach consists of translating the separate entities in the GPSS model onto separate (one or multiple) DEVS models in the target model. While performance decreases due to the introduced communication overhead, understandability is increased; domain experts can inspect the translated model and assert that it is correct, or debug it when an unexpected result is observed. We choose this second option for our translation.

As a result, each GPSS block in the source model will have an equivalent atomic or coupled DEVS model in the target model. No blocks will be lost in translation, but rather multiple blocks and connections will be added to allow for coordination of the transactions. To avoid confusion, we'll use the same naming conventions that GPSS uses. To keep in line with the GPSS syntax, this paper will refer to GPSS blocks using all caps (e.g. GENERATE, ADVANCE...). Their corresponding DEVS blocks will be written in italics (e.g. *GENERATE*, *ADVANCE*...). Operands for GPSS blocks will be provided in DEVS in a similar fashion as parameters to a class constructor. For example, "GENERATE 5" is translated to "*GENERATE(5)*". A GPSS transaction will be modeled as a DEVS event, seeing as both represent messages traveling through a model. With the exception of the *ADVANCE* block, every DEVS block will only apply its events on at most one transaction that travels through. SNAs will be modeled as function calls that yield the required value at the time of the call.

Scanning Algorithm To allow for a valid translation of GPSS to DEVS, a central *Controller* block is introduced that implements a variation of the scanning algorithm by working together with a group of *Hold* blocks, that will be placed between all sequential blocks in the translation. This means that every GPSS block will be translated to two DEVS blocks on average. Additional blocks for handling resources and obtaining statistics will also be added. Hold blocks will listen to coordination messages that the Controller sends. Whenever a transaction enters such a block, it will be stored internally until the block is told by the Controller to release a specific transaction, which is done by a *notify* event. This internal storage makes it possible for the DEVS blocks to only be part of a single transaction at a time. All necessary chains need to be present in our translation. This is done by providing the Controller block with the following four lists:

active The list of transactions that are neither blocked, delayed, or terminated. The scanning algorithm iterates over these transactions until the list is empty. The transactions on this list are ordered by priority (and time of arrival).

delayed The unordered list of transactions that are delayed until further notice. All transactions on the FEC can be found here, as well as transactions that enter a LINK.

created The list containing all transactions that are created by *GENERATE* blocks during an iteration over the *active* list. We need to make sure all *GENERATE* blocks have called their output function λ before the Controller starts with the scan. Additionally, all transactions that were delayed or blocked, but have moved, will be placed in this list, anticipating the rescan (hence, their λ needs to have been called as well).

blocked The list of transactions that cannot move to the next block due to some blocking condition. If a resource is released, the Controller will move all transactions on this list to the *created* list before rescanning. For performance reasons, the coordination of blocked transactions happens in groups instead of individually: if a single transaction of a group is blocked, the Controller can mark all remaining transactions in the group as blocked.

The Controller will notify the first transaction in the *active* list that it may move. Whenever it is to be delayed or blocked, it will be removed from the *active* list and placed on the corresponding list. When the transaction is terminated, the termination counter of the Controller is updated as required by the *TERMINATE* block and the transaction is removed from the *active* list. Because of the removal from this list, the next transaction in the *active* list will be selected for the next notification automatically. Whenever the termination counter reaches the desired value, the DEVS simulation of the GPSS model is halted.

DEVS Structure Each DEVS block that is created in the translation will get a name that is a result of name mangling. Using the *select* function of coupled DEVS, we can execute a set of translated blocks in a predefined order, which is required to implement the GPSS semantics correctly. We use the lexicographical order of the mangled name to select which DEVS blocks need to execute their δ_{int} first.

The translation of a model from GPSS to DEVS is done in three steps: first, a Controller block with name *GPSS2DEVS_2_Controller* is created, as well as a block that handles all logic switches, called *GPSS2DEVS_3_LogicSwitches*. Then, all facilities, storages, user chains, queues and tables are created with prefix *GPSS2DEVS_3_*. Last, a set of transformation rules are executed to create a DEVS structure based on the GPSS block. Each name of the translated DEVS block is prefixed with *GPSS2DEVS_5_* and each Hold block is prefixed with *GPSS2DEVS_4_*. Exceptions to this rule are the GENERATE block (which is prefixed with *GPSS2DEVS_0_* instead) and the ADVANCE and the UNLINK blocks (these latter two are prefixed with *GPSS2DEVS_1_*).

Time The *GENERATE* and *ADVANCE* blocks can cause simulation time to advance. The *GENERATE* block must halt if a transaction is blocked in the following block and the *ADVANCE* can pause a transaction (or a set thereof) for an unspecified amount of time. In the translation, an additional connection is required for *GENERATE* blocks, and *ADVANCE* blocks will continuously be notified by the Controller about which transactions must (or mustn't) be paused.

For each transaction it generates, the *GENERATE* block sets the creation time, a user-defined priority (or zero if omitted) and a unique identifier (uid) to $n \cdot i + g$, where n identifies the amount of *GENERATE* blocks in the model, g the index of the *GENERATE* block that creates the transaction and i the amount of transactions the *GENERATE* block has created already, to ensure uniqueness.

Example

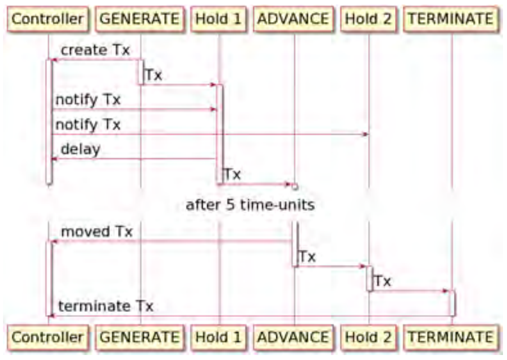


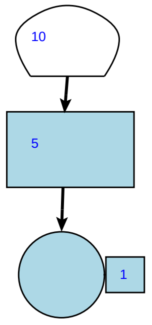
Figure 2: Sequence diagram for all messages in the translation of the example model.

Consider the GPSS model with visual concrete syntax in Figure 3a. Figure 3b shows the model in textual concrete syntax. Starting from time 10, every 10 time units, a transaction is created with a uid that follows the sequence $\{0, 1, 2...\}$. Each transaction waits for 5 time units before being destroyed. The simulation runs until the termination counter becomes zero (i.e., until a given number of messages was destroyed). Figure 3c shows the structure of the DEVS model generated from this GPSS model. Every block (except for the *TERMINATE*) is followed by a Hold block that interacts with the Controller. Furthermore, the generated transactions are sent to the Controller on the *create* port, adding them to its *created* list. The termination counter is decreased through the connection between the *TERMINATE* and the Controller. Finally, the Controller can pause the *ADVANCE* block, and it is also notified of transactions entering and leaving it. These transactions will be moved from the *active* list to the *delayed* list or from the *delayed* list to the *created* list. Figure 3d shows the generated Python(P)DEVS code for this model. For clarity, the block names are underlined in the figure.

Figure 2 shows a sequence diagram of all events that are being sent in the translated example at time 10, 20, 30, ... In the diagram, Tx identifies the transaction that is created in that time unit.

The process goes through the following phases:

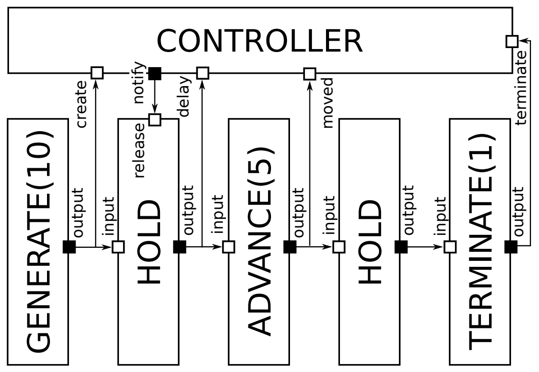
1. When the *GENERATE* block creates Tx, it sends a message to the Controller stating that the transaction has been created. Next, it will pass the message on to a Hold block.
2. The Controller notifies all Hold blocks that Tx may move. Because Tx is only in "Hold 1", it moves from there to the *ADVANCE* block, while also messaging the Controller that Tx should be moved to the *delay* list, because it's entering an *ADVANCE*.
3. After five time units, Tx is freed from the *ADVANCE* block and makes its way into the "Hold 2" block. The Controller is notified that Tx is not delayed anymore and remembers that the last notification allowed Tx to pass, so no notify is sent. "Hold 2" also remembers the previous notification and sends Tx to the *TERMINATE* block.
4. *TERMINATE* makes the Controller remove Tx from the active chain and reduce the termination counter by 1. After 5 time units, the *GENERATE* block creates a new Tx, restarting the process.



(a) Visual Notation in AToM^3.

```
GENERATE  10
ADVANCE   5
TERMINATE 1
```

(b) Textual Notation.



(c) Structure of the generated DEVS model.

```
class Model(CoupledDEVS):
  def __init__(self):
    super().__init__("Model")
    self.GPSS2DEVS_2_Controller = self.addSubModel(Controller("GPSS2DEVS_2_Controller"))
    self.GPSS2DEVS_0_L0 = self.addSubModel(GENERATE("GPSS2DEVS_0_L0", dist=dist, args=(10,),
                                        func=lambda x, t: Transaction(x, t), dt=None))
    self.GPSS2DEVS_4_L0 = self.addSubModel(Hold("GPSS2DEVS_4_L0"))
    self.connectPorts(self.GPSS2DEVS_0_L0.output, self.GPSS2DEVS_4_L0.input)
    self.connectPorts(self.GPSS2DEVS_0_L0.output, self.GPSS2DEVS_2_Controller.create)
    self.connectPorts(self.GPSS2DEVS_2_Controller.notify, self.GPSS2DEVS_4_L0.release)
    self.GPSS2DEVS_1_L1 = self.addSubModel(ADVANCE("GPSS2DEVS_1_L1", dist=dist, args=(5,)))
    self.GPSS2DEVS_4_L1 = self.addSubModel(Hold("GPSS2DEVS_4_L1"))
    self.connectPorts(self.GPSS2DEVS_1_L1.output, self.GPSS2DEVS_4_L1.input)
    self.connectPorts(self.GPSS2DEVS_4_L0.output, self.GPSS2DEVS_2_Controller.delay)
    self.connectPorts(self.GPSS2DEVS_4_L1.output, self.GPSS2DEVS_2_Controller.moved)
    self.connectPorts(self.GPSS2DEVS_2_Controller.pause, self.GPSS2DEVS_1_L1.pause)
    self.connectPorts(self.GPSS2DEVS_2_Controller.notify, self.GPSS2DEVS_4_L1.release)
    self.GPSS2DEVS_5_L2 = self.addSubModel(TERMINATE("GPSS2DEVS_5_L2", '1'))
    self.connectPorts(self.GPSS2DEVS_5_L2.output, self.GPSS2DEVS_2_Controller.terminate)
    self.connectPorts(self.GPSS2DEVS_4_L1.output, self.GPSS2DEVS_5_L2.input)
```

(d) Generated Python(P)DEVS code.

Figure 3: Translation of a simple model (time).

The "busy" section for the Controller identifies that Tx is located in the *active* list in these periods. For all other blocks, it indicates that Tx is located in the block.

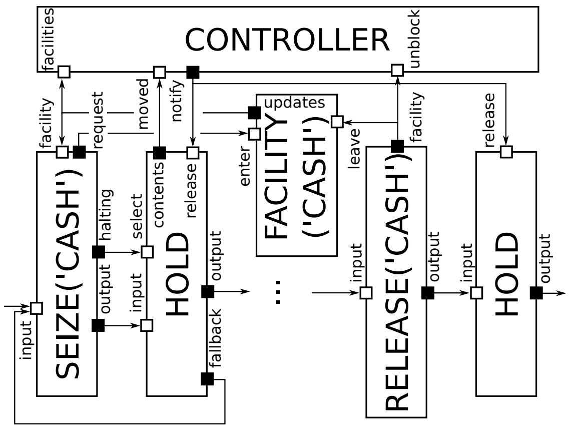
Custom Chains As far as the translation is concerned, LINK and UNLINK can be seen as a special kind of ADVANCE block. The LINK enters transactions in the chain and the UNLINK takes them out again. The delay between entry and departure of the chain depends on the correspondence between process flows, hence all transactions that are on a chain will also be on the Controller's *delay* list.

Resources In general, resources follow the same structure. Each block that gains access (be it a *SEIZE*, a *PREEMPT*, an *ENTER* or a *LOGIC*) is immediately followed by a corresponding Hold block. In fact, there is a double connection between the GPSS2DEVS-block and the Hold block: one for the passing transaction and one that indicates if the transaction should be blocked. If the Controller tells the Hold block to release a blocked transaction, the transaction retries to gain access on the GPSS2DEVS-block's *input* port. As soon as a retrying transaction is blocked, the Controller is aware that the resource cannot be accessed anymore. The block that requests access communicates with the corresponding resource whether or not access may be granted. Additionally, blocks that free up a resource also do a similar communication, besides also triggering a rescan. Furthermore, the *TEST* and *TRANSFER* blocks can be used for more complex scenarios.

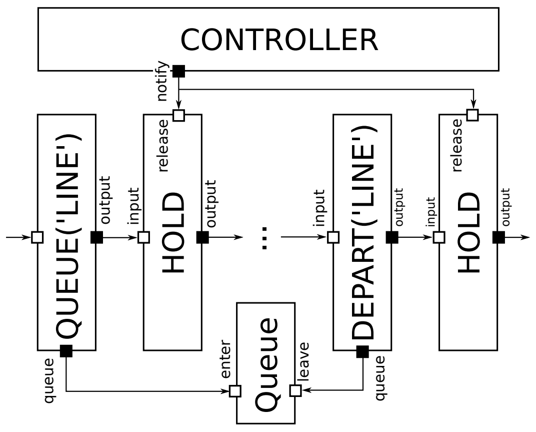
Facilities Besides the general rules that resources follow in the translation, facilities require some additional coordination with the Controller. In GPSS, a transaction has knowledge of all the facilities it belongs to. In the translation, this relationship is reversed: all facilities know which transactions have access to them, interrupted or not. This statement can be made without loss of generality. To do so, we'll have the Controller listen to facility updates. When such an update happens, the Controller must send a message to all *ADVANCE* blocks, notifying that all transactions except the last transaction that obtained the facility must be paused. Whenever a transaction releases a facility (via either a *RELEASE* or a *RETURN* block), this list is updated and another transaction will be resumed.

Logic Switches Logic switches are the odd ones out. Instead of having a single block for each switch, there is an all-knowing *GPSS2DEVS_3_LogicSwitches* block. This was done because logic switches are more often than not accessed via SNA's and because they represent a boolean state. Furthermore, instead of having a block that indicates a resource request and another that represents the release of that resource, logic switches only have a GATE block that can be blocking. For this reason, the *GATE* block implements the same general logic as, for instance, an *ENTER* and *LEAVE* combined.

Example In the example of Figure 1, we make use of a single resource: a cashier, denoted with "CASH" in the models (both in GPSS as in DEVS). Because we only have a single cashier and they can only focus on one customer at the same time, it makes sense to represent the cashier as a facility. If we ignore all but the SEIZE and RELEASE blocks from the example, we can represent the corresponding DEVS structure as shown in Figure 4a. Every facility is represented by their own block, allowing the *SEIZE* and *RELEASE* to communicate changes. Furthermore, whenever a facility is updated, it sends all changes to the Controller and all corresponding *SEIZE* (or *PREEMPT*) blocks.



(a) *SEIZE* and *RELEASE* translated to DEVS.



(b) *QUEUE* and *DEPART* translated to DEVS.

Figure 4: Translation of the basic example.

Statistics During a GPSS simulation, statistics are gathered; this section explains how queues and tables are translated. All statistics can eventually be cleared from their corresponding blocks, similar to the RESET statement in GPSS.

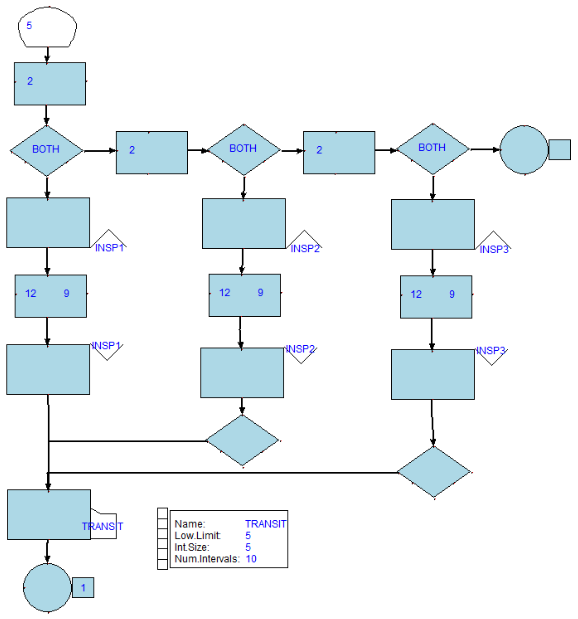
Tables We add a DEVS block, namely a *TABLE* block, for each TABLE statement in the GPSS code. Its constructor sets up the buckets to be empty and a *TABULATE* block communicates to the Table what must be tracked. Obviously, the corresponding *MARK* block is also added.

Queues Because a transaction can be in multiple queues at once and because it can be entered from multiple locations in the model, an additional atomic DEVS is used to represent the queue's datastructure. This block is called the Queue (note the difference with QUEUE and *QUEUE*). There are but two inputs to the Queue: *enter* and *leave*, who are respectively connected by the *QUEUE* and the *DEPART* blocks. Figure 4b shows how the QUEUE and DEPART blocks from the basic example (Figure 1) are translated.

Limitations The translation presented here makes a few assumptions and has some limitations, preventing its use in some contexts. Undefined behaviour that is a consequence of multiple GENERATE blocks firing at the same time should be avoided by the modeler by extensively using priorities to control the order of transactions. Because of name mangling and the name choices that were made in the translation, no facilities, storages, queues, tables and chains can have the same name, which is viable in GPSS. Furthermore, they cannot be called "*LogicSwitches*", and, because this is a proof-of-concept, the DEVS *TABLE* blocks will only track the time and the rate of arriving transactions. Additionally, (Paredis 2020) summarizes all blocks for which the translation is defined.

4 MANUFACTURING EXAMPLE

We reuse the example of a simple manufacturing shop from (Gordon 1978) to demonstrate our approach. In the example, a machine tool in a manufacturing shop produces parts at the rate of one every 5 minutes and places them on a conveyor that carries the parts to three inspectors. It takes 2 minutes to reach the first inspector; if he is free at the time the part arrives, he inspects it, otherwise, the part takes a further 2 minutes to reach the second inspector, who takes the part if he is not busy. Parts that pass the second inspector may get picked up by the third inspector, who is a further 2 minutes along the conveyor belt; otherwise parts are lost. To keep the model small, only the transit time of the parts is recorded and the details of inspectors rejecting defective parts will be ignored. Each inspector takes 12 ± 9 minutes per inspection. The model is shown (both graphically and textually) in Figure 5.



(a) Visual Notation in AToM^3.

```
*                  Manufacturing shop model 5
*                  G. Gordon Figure 11-1/9-10
       SIMULATE
 L0    GENERATE   5               ; Create parts
 L7    ADVANCE    2               ; Move to the first inspector
 L8    TRANSFER   BOTH,L5,CONV1   ; Check if first inspector is busy
 L5    SEIZE      INSP1           ; The first inspector becomes busy
 L1    ADVANCE    12,9            ; Inspect
 L9    RELEASE    INSP1           ; Free inspector 1
 TAB   TABULATE   TRANSIT         ; Tabulate parts' transit time
 ACC   TERMINATE  1               ; Accepted parts
 CONV1 ADVANCE    2               ; Move to 2nd inspector
 C2    TRANSFER   BOTH,L13,CONV2  ; Check if 2nd inspector is busy
 L13   SEIZE      INSP2           ; The 2nd inspector becomes busy
 L15   ADVANCE    12,9            ; Inspect
 L17   RELEASE    INSP2           ; Free inspector 2
 L19   TRANSFER   ,TAB            ; To tabulate
 CONV2 ADVANCE    2               ; Move to 3rd inspector
 C3    TRANSFER   BOTH,L14,TERM   ; Check if 3rd inspector is busy
 L14   SEIZE      INSP3           ; The third inspector becomes busy
 L16   ADVANCE    12,9            ; Inspect
 L18   RELEASE    INSP3           ; Free inspector 3
 L20   TRANSFER   ,TAB            ; To tabulate
 TERM  TERMINATE                  ; Remain uninspected
 TRANSIT TABLE      M1,5,5,10
       START      10,NP
       RESET
       START      10000
       END
```

(b) Textual Notation.

Figure 5: Manufacturing Example

This example is modelled in GPSS, which seems appropriate. If it were modeled in DEVS, the cognitive gap between the problem and its solution would be less intuitive, in particular for manufacturing domain experts. Table 1 summarizes the results we obtain in comparison of the TRANSIT table and the facilities between GPSS World, GPSS/H and GPSS2DEVS. We used a warm-up of 10 parts and simulated the arrival

Table 1: Obtained metrics for the example in Section 4, ran until 10 000 parts accepted.

Block	Metric	GPSS World	GPSS/H	GPSS2DEVS
TRANSIT	Mean	15.734	15.7752	15.6721
	Standard Deviation	5.479	5.4223	5.6909
INSP1	Utilization	0.831	0.834	0.8603
INSP2	Utilization	0.731	0.729	0.7419
INSP3	Utilization	0.549	0.559	0.5382

of 10 000 parts. In a deterministic scenario, there is no difference between the metrics. We notice that for this model, the generated DEVS models gives similar results to those obtained by direct simulation of the original GPSS model.

5 RELATED WORK

This work contributes to Multi-Paradigm Modeling (Mosterman and Vangheluwe 2004), which advocates explicitly modeling all relevant parts and aspects of systems at the most appropriate level(s) of abstraction, using the most appropriate modeling language(s). It is supported by modeling language engineering, including modelling language semantics as a basis for the evaluation of model properties. In (Vangheluwe 2000a), DEVS is considered as a common denominator, a semantic domain that can be used to specify the semantics of a large set of modeling languages. It introduces a Formalism Transformation Graph (FTG), a map of formalisms and transformations between them. In previous works these transformations were detailed. Both (Borland and Vangheluwe 2003) and (Shaikh and Vangheluwe 2011) provide a mapping from Statecharts to DEVS (with the former attempting a one-to-one translation similar to what is done in this paper and the latter a mapping onto a single atomic DEVS model for efficiency). In (Sanz, Urquia, and Dormido 2007) a similar mapping is implemented for SIMAN/Arena blocks by encoding their logic in a Modelica implementation of DEVS. Additionally, (Kapos, Dalakas, Nikolaidou, and Anagnostopoulos 2014) introduces a translation from SysML onto DEVS, allowing discrete event simulation of SysML models. GPSS (Schriber 1974; Gordon 1978) was introduced almost 50 years ago, but it is still relevant to this day (Ståhl 2019; Ståhl, Born, Henriksen, and Herper 2011). Many implementations of GPSS exist, such as GPSS/360 (Gould 1969), GPSS/H (Crain and Henriksen 1999), GPSS World (http://www.minutemansoftware.com/product.htm), JGPSS (Fonseca i Casas and Casanovas 2009), and aGPSS (http://agpss.com/index.html). Nowadays only the last three of this list are available in practice. DEVS (Zeigler, Praehofer, and Kim 2000) also has a lively community; the language is implemented in tools such as PythonPDEVS (Van Tendeloo and Vangheluwe 2016) and adevs (Nutaro 2015), amongst others. The relation between these two formalisms is non-trivial, since they adhere to two different world views (Overstreet and Nance 2004; Vangheluwe 2000b).

6 CONCLUSION AND FUTURE WORK

The DEVS formalism can be used as a “common denominator” to specify the semantics of many other formalisms. This paper presents a mapping of the GPSS language onto DEVS. By providing a mapping that produces an equivalent DEVS model, we benefit from the advantages that DEVS offers, including precise specification, modularity, scalable simulators, real-time execution, etc. We focused on building a working prototype for a relevant and representative subset of GPSS blocks. In the future, the translation can be extended to a larger set of GPSS blocks. The translation can also be made more usable, by providing traceability between source and target model, to support interactive debugging and testing. Several space and time optimizations are also possible, such as dependency analysis that removes unused blocks.

ACKNOWLEDGEMENTS

This work was partially funded by Flanders Make vzw, the strategic research centre for the Flemish manufacturing industry.

REFERENCES

Borland, S., and H. Vangheluwe. 2003. “Transforming Statecharts to DEVS”. In *Proceedings of the 2003 Summer Simulation Conference*, 154–159. La Jolla, California, USA.

Crain, R. C., and J. O. Henriksen. 1999. “Simulation using GPSS/H”. In *Proceedings of the 1999 Winter Simulation Conference*, 182–187. Phoenix, Arizona, USA.

de Lara, J., and H. Vangheluwe. 2002a. “AToM^3: A Tool for Multi-formalism and Meta-modelling”. In *FASE*, 174–188.

de Lara, J., and H. Vangheluwe. 2002b. “Using Meta-Modelling and Graph Grammars to process GPSS models”. In 16th *European Simulation Multi-conference (ESM)*, 100–107: SCS.

Fonseca i Casas, P., and J. Casanovas. 2009. “JGPSS, an Open Source GPSS Framework to Teach Simulation”. In *Proceedings of the 2009 Winter Simulation Conference*. Austin, Texas, USA.

Gordon, G. 1978. *System Simulation*. 2nd ed. Englewood Cliffs, New Jersey: Prentice-Hall.

Gould, R. L. 1969. “GPSS/360 – An Improved General Purpose Simulator”. *IBM Systems Journal* 8(1):16–27.

Harel, D., and B. Rumpe. 2004. “Meaningful Modeling: What’s the Semantics of “Semantics”?”. *Computer* 37(10):64–72.

Kapos, G.-D., V. Dalakas, M. Nikolaidou, and D. Anagnostopoulos. 2014. “An Integrated Framework for Automated Simulation of SysML Models Using DEVS”. *Simulation* 90(6):717–744.

Kleppe, A. 2007. “A Language Description is More than a Metamodel”. In *4^{t}h Intl. Workshop on Software Language Engineering*.

Mosterman, P. J., and H. Vangheluwe. 2004, September. “Computer Automated Multi-Paradigm Modeling: An Introduction”. *Simulation* 80(9):433–450.

Muzy, A., and J. J. Nutaro. 2005. “Algorithms for Efficient Implementations of the DEVS & DSDEVS Abstract Simulators”. In *1st Open International Conference on Modeling and Simulation*, 273–279. Clermont-Ferrand, France.

Nutaro, James J. 2015. “adevs”. http://www.ornl.gov/~1qn/adevs/. Accessed 10th May.

Overstreet, C. M., and R. E. Nance. 2004. “Characterizations and Relationships of World Views”. In *Proceedings of the 2004 Winter Simulation Conference*, 279–287. New York, USA.

Paredis, R. 2020. “Translating GPSS to DEVS”. Master’s thesis, University of Antwerp.

Sanz, V., A. Urquia, and S. Dormido. 2007. “DEVS Specification and Implementation of SIMAN Blocks Using Modelica Language”. In *Proceedings of the 2007 Winter Simulation Conference*, 2374–2374. Piscataway, New Jersey, USA.

Schriber, T. J. 1974. *Simulation Using GPSS*. New York: Wiley.

Shaikh, R., and H. Vangheluwe. 2011. “Transforming UML2.0 Class Diagrams and Statecharts to Atomic DEVS”. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation*, 205–212: SCS.

Ståhl, I. 2019. “Topics in Discrete Events Simulation For Business Students”. In *Proceedings of the 2019 Winter Simulation Conference*, 3332–3343. Piscataway, New Jersey, USA.

Ståhl, I., R. G. Born, J. O. Henriksen, and H. Herper. 2011. “GPSS 50 Years Old, but Still Young”. In *Proceedings of the 2011 Winter Simulation Conference*, 3952–3962. Phoenix, Arizona, USA.

Van Tendeloo, Y., and H. Vangheluwe. 2016. “An Overview of PythonPDEVS”. In *JDF 2016*, 59–66: Cépaduès.

Vangheluwe, H. 2000a, 08. “DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling”. *IEEE International Symposium on Computer-Aided Control System Design*.

Vangheluwe, H. 2000b. *Multi-Formalism Modelling and Simulation*. Ph. D. thesis, Universiteit Gent.

Zeigler, B. P. 1976. *Theory of Modelling and Simulation*. New York: Wiley.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. New York: Academic Press.

AUTHOR BIOGRAPHIES

RANDY PAREDIS developed GPSS2DEVS as part of his Masters thesis in the Modeling, Simulation and Design Lab (MSDL) at the University of Antwerp (Belgium), where he is currently a PhD student. His e-mail address is Randy.Paredis@student.uantwerpen.be.

SIMON VAN MIERLO is a post-doctoral researcher in the MSDL. His PhD thesis developed debugging techniques for a variety of modeling and simulation formalisms, including DEVS, by explicitly modeling their operational semantics using Statecharts. His e-mail address is simon.vanmierlo@uantwerpen.be.

HANS VANGHELUWE is a Professor and head of the MSDL. He has a long-standing interest in the DEVS formalism and is a contributor to the DEVS community of fundamental and technical research results. Since he first learned GPSS in the late 1980s, he has admired its conciseness and elegance. His e-mail address is hans.vangheluwe@uantwerpen.be.